Paper Summaries

Petros Maniatis

April 22, 2004

1 Categories

1.1 Mobility

Virtual Internet Protocol [169], [167], [168].

Internet Mobile Host Protocol [136], [137], [124]

Cho-Marshall [50], [49]

Columbia [90]

Coda [122]. [97] for the file system.

1.1.1 Personal Mobility

MPA [11].

Ninja/ICEBERG/Universal-Inbox

TOPS [9].

Personal Identity [?] for a general overview of IDI. [?] for the current Internet draft of the associated protocols. [53] for Microsoft's Digital Nervous System approach.

1.2 P2P Infrastructure

A first pass at an intentional name service and discovery protocol in [1].

Security A "trust-but-verify" proposal for a robust DOLR using hop-by-hop quorum mechanisms during forwarding and for structure management [12].

1.3 Network Connectivity Management

Self tuning in [114].

Quasar [88], [89]

Characteristics Discovery [?](see page ??).

Mobiware [10].

1.4 Wireless

[161] on vertical hand-offs, [37] on performance of TCP. [70] for distillation to facilitate wireless clients.

1.5 Operating Systems - General

Words of Wisdom [107] on categorization and experiences.

System Overviews [155] on VM/370.

1.6 Extensibility

A motivational/inspirational paper (particular on databases though) is [164].

Synthesized [68] on component-based OSes.

Exokernels [65] for a general description. [95] for disk multiplexing. [24] for SPIN.

Microkernels [46] on V++.
[80] on KeyKOS.

Virtual Machines [33] for Disco.

1.6.1 Infrastructure

[174] on service infrastructure extensibility (via "active names").

1.7 Security

[59] is an early, general survey. Protection Rings in [154]. The authoritative, seminal paper on authentication algorithms in [127]. An attack against the Needham protocols in [112]. [109] presents a model for calculating the appropriate cryptographic key size, given certain constraints. [178] on how easy it is for normal people to use security applications?

Kerberos [160] for a position. [**?**] for its limitations (see page **??**).

DDSSA Presentation of the DEC secure system infrastructure for the DDSSA system (and associated theory) in [105].

Protocol Design [36] describes a logic of authentication.

Global Trust A paper-form global CA in [5]. Betsi [151] is a system for distributing trust in disseminated software.

Threat Models Ballardie lists multicast-specific security threats and some counter measures [14]. Anderson describes the need for more comprehensive threat models that encompass all components of a secure system including its human operators [4].

Audit Trails SNIF [2] collects logs from individual hosts in a LAN and allows an operator to browse through them. NERD [156] is a similar set of tools that operates in parallel with syslogd. ASAX [121] is a rule-based audit-trail analysis tool.

Denial of service Needham identifies the salient points of DoS [126]. SOS [96] protects communications within a known population of end-points (users or IP-addresses) against DoS interference on the Internet by provisioning an overlay among core or ISP routers. Generalization of SOS and some "intelligent" attacks against it [180].

Services The first cryptographic file system [29].

1.8 Fault Tolerance, Reliability and Transactions

[75]. Rio file cache in [44]. Quicksilver, a transaction-based distributed system in [?]. The big database picture on top of an OS in [?]. Tandem NonStop in [16]. [110] for recovery and replication in Harp. Fault containment in [42] for Hive.

1.9 Virtual Memory

Multics: [?] for the "hardware" side. [54] for the software side. [165] for large page tables. [82] for external page-cache managers.

1.10 Distributed Operating Systems

[?] on LOCUS. [132] on the Information Bus. [27, 153] on Grapevine.

RPC and relevant info [26] on Modula-3 network objects.

CORBA [175] on CORBA, generally. [?] is a book on it (see page **??**). [140] is a later reference guide.

SPRING [117] for an overview. For the name service, [142] and page 176, [?] and page **??**, [?] and page **??**. Persistence in [?] and page **??**. Subcontracting in [?] and page **??**.

1.10.1 Resource Location

[134] on host anycasting, useful for resource location or intention-based routing.

1.10.2 Replication

[93] on the hierarchical Pleiades approach and [102] on the online version of the same technique.

1.10.3 Naming

Organization [104] is one of the first attacks on Global Naming. [47] talks about the decentralization of a global naming service. [119] Talks about the birth of the DNS and [55] describes some DNS traffic analysis.

Philosophy [131] is an old short thesis on content-based naming. [1] on the Intentional Naming System. [130] is a short thesis on the dangers of silly naming systems.

X.500 [91] on the X.500 directory service. [129] on creating a "descriptive name system" on top of X.500 (or any hierarchical name system, for that matter). [79] describes a way to implement guessable names within X.500.

Personal [139] on the Profile Naming Service.

1.11 Multiprocessors

[7] on scheduler activations. [42] on fault containment.

1.12 Process Communication and Synchronization

Deadlocks in [51].

Message Passing [17] capabilities/messages in DEMOS. Synthesis in [114]. RPC, PARC style in [28].

Monitors and Locks Monitors, the Hoare approach in [84]. Non-blocking locks in [22].

1.13 Filesystems

Basic [146] the original UNIX filesystem. [116] on the Fast File System for UNIX. [148] on log-structured on SPRITE.

Extensible [98] for v-nodes. [149] for their evolution.

Distributed [81] on the Zebra filesystem. [**?**] on serverless filesystems. [97] for CODA. [110] for Harp. [152] on the original NFS design. [101] for VAXclusters. Weighted-voting in replication in [74].

Input/Output [66] on the first inklings of device drivers, device-independent I/O interfaces, standard input/output, stream redirection. [158] on a survey of disk optimization techniques.

1.14 Networking - General

[162].

SNMP [**?**] on SNMP MIBs (see page **??**).

1.14.1 Active Networks

[166] for a survey of '97 active network research. [174] on the active names spin on mobile code.

1.14.2 Multicast

Core-based trees [15] replace per-source shortest-path trees for IP multicast. [14] lists security threats specific to multicast. [**?**] provides a mechanism for performing access control on a multicast stream at line speed based on cryptographic bin classifiers.

Dissemination of software can be secured using Betsi [151].

1.14.3 Services

Using admission control to protect web servers from overload: session-based [48]. On-line optimization of admission control and scheduling parameters [38].

1.15 Personal Communication

[27] on the person-to-person messaging system in Grapevine. The extended individual in [**?**]. Person-level routing in [150]. The current, as of November 1999, IETF IMPP working group model [**?**] and protocol requirements [**?**]. The IETF identity infrastructure in [**?**]. A blanket survey article in [62].

1.16 Administration

Manuals [135]

1.17 Computational Geometry

[21] from the CS368 class.

1.18 Artificial Intelligence in Systems

[71] on automatic model inference (decision trees for compression).

1.19 Conferences

2 The design and implementation of an intentional naming system (William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan and Jeremy Lilley)

[1]

INS is a resource discovery system for mobile networks, based on attribute-value pair "names". Names describe *what* is sought, not *where* the sought object lives. Message routing and name resolution are bound together (the authors call this "late binding"), so that physical name mappings can change even during a long-lived session. However, reliable delivery cannot be guaranteed with late binding.

The expected mobility can come in different flavors:

- the sought service changes network address,
- the sought service changes network (i.e., from ethernet it moves onto a wireless network),
- the characteristics of the sought service change (i.e., its performance goes down or the network path to it becomes congested).

2.0.1 The names

Example abstract names one can resolve in this scheme are: "the closest geographically, least loaded printer", "the least loaded fileserver with the best network performance to where I am", "all mobile cameras in section A of this building".

Names belong to an attribute hierarchy. The nodes of the hierarchy are not attributes; they are attribute-value pairs. This allows the children of a node to change structure, depending on the value of the parent. A good example from the paper is that "country=us" has a child "state=CA", but "country=canada" has a child "province=BC". If we, instead, had hierarchies of attributes and values, under country we'd have to deal with both provinces and states.

2.0.2 The name resolvers

Intentional Name Resolvers are both name resolvers and message routers (since, in INS, messages piggyback data with the intentional name of their destination). Services attach themselves on an INR, which advertizes the attribute-value characteristics of its attached services to its neighboring INRs.

Delivery to names can be one of the following:

- *intentional anycast*, or single-end picked among those satisfying the intention of the name and optimized for an application-controlled cost function, or
- *intentional multicast*, i.e., similar to multicast, where group membership is determined by whether nodes satisfy the intention of a name.
- *early binding*, i.e., single-step intentional resolution, where the local name resolvers just returns the network addresses of those servers it thinks best fit the intention.

The first two options belong to the "late binding" technique, where name resolution and forwarding are done by the naming infrastructure. The third option separates forwarding from naming.

For example, the name "color printer" would resolve to the closest, fastest, highest-resolution color printer to where I am (if my cost function is proximity, speed, resolution) in "intentional anycast"; in "intentional multicast" the name would resolve to all color printers I can reach. In early binding,

it would just resolve to the network addresses (IP addresses) of the color printers I know are out there, along with their metrics, as I have them in my cache.

INRs organize themselves into a (not necessarily minimum) spanning tree, using average delay (round-trip time) as the cost metric. Every new INR in the INR cloud pings all others to figure out which currently active INR is the closest one (using the delay metric). It picks that one as its next hop neighbor. The tree is not minimum and it's not robust (it has many single points of failure).

Periodic system-wide updates take the form of flooding. A new name-specifier is made known, along with its associated IP address, port number and transport type, a routing metric, the next hop INR along with its delay metric and a unique identifier of the announcer.

The list of currently running INRs is maintained and provided by a central *Domain Space Resolver*.

2.0.3 The clients

Clients can either request name resolution (along with forwarding, in the case of late binding), or discovery. Clients themselves also have intentional names (so that services can send them their results).

Services pick the metrics on which they wish to receive routing (for instance, their average load, or their print resolution). They also set an announcer ID for themselves (to differentiate them from other similar services running on the same node). The ID is a concatenation of the IP address with the startup time of the server process.

2.0.4 Design goals and evaluation

The main design goals addressed are:

- Expressiveness
- Responsiveness (i.e., quick adaptation to topology or quality changes)
- Robustness
- Easy configuration (i.e., least manual effort)

Performance is significantly hurt by the CPU-intensive name resolution and update processes. To alleviate the problem, partitioning services into different *virtual spaces* has been proposed and tried.

2.0.5 Commentary

In p. 188, fourth paragraph, why is forwarding the message to the "best" end node so much better and dynamic than early binding? Is this to save against changes effected between the time of the resolution and the actual use of the return network addresses?

In p. 189, figure 1, the service announcing an intentional name is in the middle right of the figure, not the bottom right.

In p. 190, last paragraph. Why do we need the routing tables? We already have routing tables at the IP level. Why do we route through INRs? Perhaps to catch as yet unsettled updates?

There's no information on how a client finds and/or picks an INR.

It's not clear how v-spaces remain balanced.

The performance evaluation is rather superficial. Arbitrary distributions are assumed for the growth of name trees and the behavior of virtual spaces.

It is not clear why late binding is useful. If the first-hop INR is rapidly updated, which is what the architecture strives to accomplish, then why should the system go through the INR network to deliver the data?

3 An Overview of SNIF: A Tool for Surveying Network Information Flow (J. Alves-Foss)

[2]

SNIF is a tool for collecting network monitoring data at the individual clients of particular networks, which can then be collated and monitored at a central location. Each client runs filters that can be user-configurable, and a logger that puts the stream of each filter into a single log that it transmits to the remote central monitoring station. At the central monitoring station, a collator combines all input streams from individual clients and stores them in a database. Finally, the monitor is the interface with a human

operator; it offers selection facilities to present different views of the data, and triggers/alarms for specific exceptional conditions,

4 Persistent Authenticated Dictionaries and Their Applications (Aris Anagnostopoulos, Michael T. Goodrich, Roberto Tamassia)

[3]

This paper presents a introduction to the concept of persistent authenticated dictionaries. It describes the design of two such dictionaries, one based on red-black trees, and one based on skip lists. Finally, it presents measurements of an implementation of persistent authenticated skip lists compared to ephemeral (non-persistent) skip lists and red-black trees.

The main motivation for authenticated dictionaries is the secure distribution of large amounts of information by untrusted intermediaries (directories). The source of the information need not be online while the distribution to clients takes place; the source merely produces the dictionary, signs a digest for that dictionary, and distributes the dictionary and the digest to the intermediaries, who in turn make the information available to clients.

The canonical example of authenticated dictionaries appears in certificate revocation services: the certification authority produces signed lists of revoked certificates, and distributes them to untrusted directory servers. Clients check with the directories whether a certificate is still valid or revoked; the directory returns an answer (yes or no) and a proof linking this answer to the signed digest of the revocation database. Clients who can successfully verify the proof given the signed digest of the database can believe the answer of the untrusted directory.

The authors describe the additional problem of maintaining persistent authenticated dictionaries. These can not only answer the question "is element e in the authenticated dictionary", but also the question "was element e in the authenticated dictionary at time t".

They construct persistent authenticated directories in ways similar to the construction of persistent directories (without the authentication information). A new root of the directory is maintained for each new snapshot of the directory. Only nodes in previous snapshots that change in the new snapshot are copied across snapshots. For example, a persistent insertion into a

red-black tree creates new copies of all nodes on the path from the root to the location of the newly inserted node.

Issues This implementation of persistent authenticated red-black trees maintains the stored data at the leaves of the tree, instead of in internal nodes as well, as in authenticated search trees [**?**]. This creates twice as many nodes as in authenticated search trees, although asymptotically the two schemes are equivalent. In very large dictionaries this is a significant overhead.

The authors don't give any performance information about their persistent authenticated red-black trees. They maintain that persistent skip lists do a better job, based on evidence that they are more efficient than red-black trees in the ephemeral case.

Finally, the scheme presented here relies on the source to maintain a consistent, tamper-proof history of its updates: the source calls a set of updates t_i and signs the statement. Beyond that, given time stamps t_i and t_j, where $i < j$, the source provides no proof that a snapshot timed at t_i was created before a snapshot timed at t_j.

5 Why Cryptosystems Fail (Ross Anderson)

[4]

In this paper, the author makes a case for revisiting the paradigm under which the security of critical systems is protected. Researchers and engineers, the author claims, work on preventing what is *possible*, instead of focusing on what is *likely*; they work on stopping sophisticated cryptanalytic attacks against security subsystems, while ignoring social, implementational, or deployment-specific vulnerabilities. Part of the problem is rooted in the reluctance of governments, banks, and such high-security organizations to reveal the manner in which their secure systems are attacked, with or without success. In contrast, in high-assurance, safety-critical environments such as aviations, failures are highly publicized and extremely important in designing the next iteration of the previously faulty system, both from the engineering standpoint, but also in terms of the skill-sets of those operating the relevant equipment.

The paper proceeds to identify methodically simple, complex and fairly sophisticated examples of attacks against the banking system and, in specific, against the automatic teller machines of banks. Statistics show that the

overwhelming majority of incidents consist of low-tech attacks perpetrated via weaknesses in operators' service quality or moral fiber.

The author recommends that secure systems certifications include not only the secure components involved in the operation of a complete business process, but also the acquisition and maintenance of the skills necessary for the humans involved in the process to conduct their roles adequately, as well as the complete consideration of the failure modes of every component, secure or other, involved in the process.

6 Serverless Network File Systems (Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, Randolph Y. Wang)

[8]

The general idea is: ZebraFS [81] completely decentralized. Any participant can take over the role of any failed component. Also, xFS employs cooperative caching, to use participant's memory as a global file cache.

Major assumption: a cluster of cooperating workstations (NoW), whose kernels trust each other completely. A cool application of this is using a NoW as a fast NFS server.

Three main problems solved:

- scalable, distributed metadata; scalable, distributed cache consistency management, reconfigurability after failure
- scalable subgrouping to avoid excessive distribution
- scalable log cleaning

Metadata are handled by managers. A shared, globally replicated manager map assigns a metadata manager to a file. The assignment of files to managers is done through assigning different bit areas of the inumber to different managers. New files are given appropriate index numbers so they are managed by a manager living on the same node that created them.

Each metadata manager has an IMAP to locate the blocks to its files. In fact, an IMAP entry contains the disk location of the file's i-node, which contains the block addresses or the addresses of indirect blocks.

Storage servers are split into stripe groups. A stripe is split among fewer storage servers than all those available, to make sure that stripe fragments don't turn too small or, conversely, that clients don't have to buffer enormous amounts of data. Each stripe group has its own parity. Therefore, simultaneous storage server failures in different groups can be recovered from. Also, clients can write to different stripe groups at full speed at the same time.

i-nodes are only cached at the managers, so that the manager doesn't get bypassed when a local cache can't satisfy a request. This was decided so that cooperative caching could always be preferred to disk access. Caching is done per file block, not per file and follows the write-invalidate protocol.

Recovery is accomplished through checkpointing, roll-forward and distributed consensus for membership list recovery.

7 The Eternity Service (Ross J. Anderson)

[6]

The main drive behind this paper was the design of a storage system that displays properties similar to the design goals of the Internet: resistance to denial of service attacks through redundancy and scattering.

The service conceptualized here, *the Eternity Service* aims to reinstate the irreversibility of traditional publishing to electronic publishing, the process of placing an electronic document in a publicly accessible digital medium. The overarching directive is: maintain availability of published information, even in the face of locally all-powerful attackers, who cannot however bring the entire Internet down.

The author points to the earliest identification of *anonymity* as the strongest force against selective denial-of-service attack in the literature([**?**]). If requests for data are anonymous, and so are data stores, then no legal force can stop those data from being distributed: there is nowhere to serve a cease-and-desist order.

The functional design envisioned for the Eternity Service is simple: a digital document is uploaded into the service, accompanied by the duration of intended storage (say 50 years) and adequate payment, in the form of a digital coin.

Copies of the document are distributed on cooperating servers around the world which only serve a common protocol – not a common manager, since a common manager could be coerced or tricked into renegging on the

service guarantees. Furthermore, once the document has been inserted into the service, it cannot be removed, not even by its owner, which makes "unpublishing it" an impossible task. For example, during initial insertion, 100 servers are chosen to keep the document, of which only 10 are remembered in random (to spot-check the contract enforcement). If a terrorist attempts to "convince" the owner to remove his document for the service, all the owner can do is produce the list of 10% of the storage servers holding the document.

One trick against legally coerced de-publication identified here is the *perjury trap*. A simplistic instance of a perjury trap is to include in the login procedure a requirement that the agent declare under oath that he is a free agent; not many courts would compel people to commit perjury.

Indexing of stored information is as important as the stored information itself. If a user cannot find a way to retrieve the needed data, then the data is almost as good as gone. An approach would be to insert indices or directories of eternal documents into the Eternity Service as well, starting from a few well-known roots.

The practical issues with payment are also quite important. It must be made easy for banks to deal with anonymous payments with regards to anonymous documents; otherwise, banks can be coerced to not cash digital coins, which can coerce servers to not store a document long enough. On the flip side, it must be easy for payments to be conditional on the server's continued storage of the document; i.e., a server shouldn't be allowed to collect "rent" on a file it no longer stores. This can be accomplished by having banks require a challenge-response interaction with a server, verifying possession of the entire document, before an instalment of the service charge is paid.

Finally, time is essential to the deployment of such a service. If current secure time methods, such as GPS, are used, then those controlling these methods (i.e., the US Department of Defence in the case of GPS) essentially control the Eternity service. Issues involving time must be resolved for such a service to become robust beyond the reach of any single government.

8 TOPS: An Architecture for Telephony Over Packet Networks (N. Anerousis and R. Gopalakrishnan and C.R. Kalmanek and A.E. Kaplan and W.T. Marshall and P.P. Mishra and P.Z. Onufryk and K.K. Ramakrishnan and C.J. Sreenan)

[9]

Integration of heterogeneous networks at the application level.

Target application is telephony (and telephony-related services, like voice mail). The main contribution is an architecture for connecting telphony networks with the Internet, and the associated protocols for using packet-based transports to carry logical channels.

It's main components are:

A Directory Service Its main goal is to associate a "distinguished name" (a unique ID for a person) with a set of "call appearances" (application-specific profiles, including addressing info, terminal capabilities, authentication keys etc.) and a set of "query handling profiles" (a set of matching rules, determining which call appearances will be returned to a particular caller). Each person has a "home directory", which stores an uncacheable authoritative version of his information. The home directory is located using an inter-server protocol.

A Terminal Tracking Service Its main goal is to be the "Home Agent" in IP-based or other networks. It pretty much tracks terminals (what we call hosts in CS-land) when they change network addresses. There's no concept of a foreign network, though. When a terminal moves into the jurisdiction of another TTS, the directory entry is changed to reflect the move and the old TTS becomes yesterday's news (except for forwarding retarded packets).

Gateways Galore To bridge the gap between packet networks and the PSTN.

An Application Layer Signaling protocol This deals with multiplexing a single communications pipe among multiple data flows. Many logical

channels can live on top of a single transport connection and ALS is responsible for helping that happen. It also deals with call establishment and tear-down.

An abstraction protocol for Logical Channels Here the telecommunications world discovers the concept of multiplexing. The whole idea of having independent flows of data (or control) over a single transport-level connection is discussed.

A set of conferencing mechanisms A very important aspect of any conferencing mechanism is how it deals with echo cancellation. This echo is caused by the fact that output at one of the participating stations is "overheard" by the microphone, reinserting it into the conference. The net effect is that audio tends to echo multiple times in the conference, until it eventually dies out. TOPS supports both centralized mixing (which means, there's can either be a central echo server that mixes individual signals and redistributes them, making sure the appropriate signal has been cancelled for each recipient, or the cancellation occurs at each end host).

9 Robust Distributed Name Service (B. Awerbuch and C. Scheideler)

[12]

This paper proposes a philosophy and a particular proposal following that philosophy for implementing a robust lookup service in a loosely coupled distributed system roughly arranged as a DHT or DOLR overlay.

The philosophy ("trust but verify") suggests that peers view the overlay as a collection of coarse-grained "regions" within the identity space. A peer robustly performs overlay operations via a quorum (i.e., voting) mechanism within those regions that it deems "safe," where "safe" is rigorously defined as a region with a given size (in contained consecutive identifiers from the identifier space) and given number of therein contained peer neighbors. If such regions can be constructed and maintained in such a way that a region quorum contains at least one honest peer, then any operation that proceeds over a series of such regions can be expected to complete successfully.

The authors assume a context in which honest peers remain honest forever (i.e., cannot be taken over), the number of adversary peers are at all times limited as a fixed proportion of the size of the entire population (though adversary identities may be infinite), communication is reliable and immune to message suppression or alteration, bandwidth is unbounded, peer identities (e.g., IP addresses, not overlay identifiers) are certified (i.e., identity spoofing is impossible), peer arrival and departure rates are bounded.

The specific trust-but-verify proposal described is similar to typical ring-based DHT constructs. A peer maintains connections to peers within a safe region around its own overlay identifier (computed for example as ID=h(peer name)) as well as within safe regions around the overlay identifiers at exponentially decreasing distances (in the identifier space) from the identifier of the peer (i.e., ID+/-half way across the identifier ring, ID+/- a quarter of the way across the ring, etc.). Then forwarding of messages from a source to its destination overlay identifier proceeds hop-by-hop over such "finger" regions (the authors do not use the "finger" term). However, each outbound hop is sent to all connected peers within the target regions, whereas inbound hops are forwarded only if received by at least half of the recipient's known connected peers within the previous-hop region. Note that region boundaries are dependent on the identity space structure, so regions maintained by different peers are either distinct or one subset of the other.

The concept of a safe region is based on a balls-and-bins argument on the distribution of honest peers in regions. The premise makes sense, since honest peers are expected to have random overlay identifiers, whereas dishonest peers are expected not to. The argument proceeds by employing a distribution tail lower bound to support that with high probability, an honest peer with connections within a region will connect to at least a minimum number of honest peers within the region. Given that peers connect to no more than a maximum number of peers within such safe regions, the argument proceeds to estimate the maximum number of adversarial peers to which an honest peer connects within a safe region.

Questions: it is unclear how the system degrades when the limit on adversarial presence is broken. Given how easy it is to obtain multiple identifiers and assign them arbitrary overlay identifiers, it is not clear how such an assumption on adversarial presence can be reasonable. Once this assumption is broken, no quorum mechanism can, by itself, provide immunity from adversarial action.

10 The Eternity Service (Back, Adam)

[13]

This paper describes one of the implementations of the Eternity Service [6]. Its goal to to create "a distributed data-haven" and implements it on top of USENET. The USENET, and in particular the alt.* newsgroups, is a case where distribution across multiple jurisdictions results in a resilient overall-service: although many well-funded attackers have tried to remove individual groups under the alt.* hierarchy, they all have failed, because there are just so many USENET administrators with individual opinions and diverse commercial interests.

This implementation allows access to a stream of encrypted USENET news articles as a web-page with a persistent URL. These persistent URLs are assigned within DNS-like domains, under the non-existent top-level domain "eternity." Names under that are given on a first-come, first-serve basis and are maintained as long as the referrenced document remains. Hierarchical association through the hostname or file space within a host name does not mean any association between those owning the names. For example, "http://bluebox.eternity/" is neither associated with "http://my.bluebox.eternity/", nor with "http://bluebox.eternity/blackbeard".

Eternity servers know which documents they house. You can ask them wild-card questions (something like, show me all documents with the name pattern "http://*.eternity/"). This capability can be turned off.

The process to construct a new anonymously distributed page is as follows:

- Create a pseudonym for cryptographic use (i.e., including a public/private key pair)
- Create the document
- Optionally encrypt, sign and/or compress the document
- Decide on the distribution parameters: the Eternity URL for the document (e.g., "http://useless.eternity/really.html), its caching characteristics (cachable, uncachable, or cachable in an encrypted form), whether it should be listed in directories or not and a textual description
- The ascii representation of the document, of its distribution parameters

and, optionally, of the public key used on the document, as created in the first step, are all put together into a single file

- The final submission document is encrypted symmetrically, using as the key the hashed URL. Unless a reader knows this URL, he cannot read the document
- The encrypted submission document is submitted to a news server through one of the many available anonymous remailers.

This paper is just an introduction to an implementation of the Eternity service and doesn't go much further than describing this implementation.

11 Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing (Tony Ballardie, Paul Francis, and Jon Crowcroft)

[15]

Core Based Trees were a response to the then prevalent IP multicast architecture. That architecture builds a per-source shortest-path spanning tree to every member of the multicast group. This is done either via the use of the distance vector multicast routing protocol (DVMRP), which floods a group advertisement throughout the network, expecting prune message from routers whose downstream network is uninterested; or various link-state protocols that propagate information on which groups are interesting to the end-points of particular links, allowing each sender to receive global information he can use to compute his own shortest-path spanning tree.

The authors argue that neither technique would scale to large numbers of groups over the entire Internet, and propose instead a multicast environment in which a single tree is built per group based on reverse-path forwarding. Only routers on the path from interested clients to the "core" of the tree (who can be the initiator of the group or a designated core router) join the tree and remember any group-specific state. Senders route their requests towards the core; when a router on the CBT intercepts such requests, it propagates them to every other router along the tree links.

CBTs may be suboptimal in the actual paths taken by packets from individual senders to every recipient, and form a single-point of failure at the core. Such problems can be overcome, according to the authors, using dynamic core trees, and multiple cores for the same tree.

12 Multicast-Specific Security Threats and Counter-Measures (Tony Ballardie and Jon Crowcroft)

[14]

This paper looks at why multicast communication is a tough environment to secure. It suggests that since traffic necessarily cannot be two-way between individual recipients of the multicast stream and its source, traditional security solutions to the problems of authorization, authentication, and confidentiality are more expensive (e.g., signing each packet), unworkable (e.g., using cheaper MACs for stream integrity), or "unfair" (e.g., using an expensive security mechanism, impacting everyone's performance, when not all recipients require it). IP multicast addresses are easy to guess and offer little group membership control, allowing attackers to pose as group members and receive anything sent to the group. Similarly, the lack of access control at the group membership level means that anyone can send anything to the group as a whole. Finally, flow control is harder, making congestion caused by multicast harder to prevent.

The authors identify security threats against multicast that are either passive (traffic observation/eavesdropping) or active (denial of service). More particular threats include masquerading (i.e., spoofing) for control traffic such as group joining or for data traffic, replay of packets that can leverage the multiplicative effects of multicast to replicate massively, and repudiation of sent traffice that is seen by many recipients.

The authors argue in favor of public key cryptography for multicast environments, since symmetric cryptography tends to be very expensive for one-to-many communication when some recipients may be malicious. As a result, the authors assume a widely deployed public key infrastructure for all of their countermeasures.

To provide group membership access control, the authors suggest the use of a core-based-tree [15] of authorization servers (ASes). The group initiator, who has the ultimate control over the membership of the group, issues a

group access control list, at the granularity of IP subnets (i.e., IP address and subnet mask pairs), which it transmits to all ASes.

When a client wishes to join, it sends an appropriate request towards the core of the AS tree. The request is intercepted by the nearest AS, which checks the request against the access control list of the group. If the request is permissible, the AS issues an authorization stamp that acts as a capability for the client's join operation. The client can then forward that capability along with a join request to its local router, which presents this capability to its upstream router on the way to the actual multicast group's CBT.

13 A NonStop Kernel (Joel F. Bartlett)

[16]

Wow! I think I just read an early description of a Remote Procedure Call, before it was called that! There is an interesting approach to canceling a request or a response, though.

The interprocessor protocol looks a lot like an early TCP clone. No slow start or congestion avoidance stuff is there, obviously, since the interprocessor buses are fairly reliable (less than one fault in a month reported). Processors broadcast a heartbeat every second. If two successive heartbeats are skipped by a processor (and detected by others, obviously), the processor is deemed dead.

The concept of process pairs is also brought up. Every process is backed by a checkpoint process, which receives status updated after every message operation of the primary. If the primary process dies, the backup becomes the primary and picks up where the dead one left off. To deal with non-idempotent requests handled by dead servers, a buffer of responses is retained, so that already answered requests can be answered again without recomputing a response.

14 Limitations of the Kerberos Authentication System (Bellovin, Steven M. and Merrit, Michael)

[18]

Flaws deriving from the Project Athena environment.

- Kerberos prior to version 5 was a user-to-server protocol. It could not be used as a host-to-host or user-to-user protocol.
- On multi-user machines, session keys are stored in files which can be compromised (used without permission), if access control to the machine is subverted. If I gain root access, I can use somebody else's session key (if it's still available/current)
- Keys cannot be stored on /tmp, if it is a remote file system. In that case, they are blatantly sent out over the net (BAD!)
- Tickets are bound to IP addresses. What happens with multi-homed machines? *What about mobility?*

Protocol Flaws

Replay attacks Supposedly averted by authenticators. They have a very short lifetime (typically 5 minutes). However, it is fairly easy to snoop an authenticator and reuse it within its lifetime.

The only solution would be authenticator caching, to allow for replays. That hasn't been implemented easily yet. For TCP, it's very difficult, since servers are normally spawned by the main listener. Only threading would work. For UDP, it would present problems with application-level retransmissions. If the application could reconstruct authenticators for each retransmission, the problem might be obviated.

Time Service Kerberos assumes that there exists an underlying authenticated time service that can be used. This is not true however, not to mention that even authenticated time servers could be subverted, off wire (by using a fake radio transmitter, for instance). Instead, challenge/response sounds like a better approach, proving (using a nonce) that each party in fact possesses the session key. UDP would still be hard though (would need to retain state per use).

Passwords It's relatively easy to exploit user weaknesses in choosing passwords to make educated guesses. Passwords alone are not a good idea to determine a client's key. Diffie/Hellman could be significantly better.

Login spoofing The same problem applies elsewhere as well. However, a good solution for other cases (e.g.,. One Time Passwords) cannot be applied to Kerberos, since an initial request is always encrypted by Kc.

Inter-session chosen plaintext attacks Message structure should be revised to prevent known-plaintext attacks.

Session keys They shouldn't be multi-session keys. They expose the system to cryptanalysis.

Ticket scope Ticket forwarding imposes problems, since it assumes equal levels of trust among all sites participating in the forwarding operation. Furthermore, multi-realm authentication can have problems, especially in routing, since there is not global realm name service.

Hardware Design

- To perform all key operations (and prevent keys from appearing in hosts' main memories) a Kerberos-aware secure cryptographic box could be used. This should not transmit keys in any way. I should only perform cryptographic operations.
- A keystore, a secure storage facility, could be used to maintain key databases, without understanding the protocol.

Certain portions of the Kerberos Protocol do not belong there, namely, message encoding (something like ASN.1 or BER should be used instead) and encryption mechanisms (let them be pluggable - the protocol should be correct for any encryption mechanism).

Proposed changes:

- challenge/response instead of time-based authentication
- standard message encoding
- don't use passwords at the login. Use a one-way function of the passwords given a random number instead (requires some sort of handheld device to compute that)
- separate the encryption layer.

- use multi-session keys to provide true session keys
- support special-purpose hardware for cryptographic operations and secure storage
- do not distribute tickets for users - authenticate initial exchange
- support optional extensions, like Diffie/Hellman.

An extremely interesting paper!

15 Efficient broadcast time-stamping (Benaloh, Josh and de Mare, Michael)

[19]

This paper proposes an extension/refinement to the time-stamping solutions described by [77], to remove the need for a central authority, substantial cooperation among participants for posting and validation of time-stamps, and substantial amounts of storage.

This proposal makes two major changes in the previous approach:

- Documents are groupped in batches, called *rounds*. Rounds are distinct and have a full ordering among them. Documents within a round, however, have no ordering among them. The time granularity within a stamp is therefore that of a round (usually, a day).
- Cooperation of participants is effected over an authenticated broadcast channel. Participants do not need to coordinate with each other.

The authors present two distinct schemes: a flat scheme with no central authority, and one which imposes a hierarchical organization upon participants.

Both schemes assume that:

- All participants are known in advance and have a unique identifier known by all
- All participants can broadcast authenticated information
- All broadcasts can be received by all participants

15.1 A flat decentralized scheme

In each round (e.g., every day) the fingerprints of all documents that need to be stamped are broadcast. Starting from the previous round's time stamp, all broadcast fingerprints are chained in the order of their owner's identifiers (an arbitrary order known to all). The value produced at the last hash chain link becomes this round's time stamp, is agreed upon among all participants and gets published (e.g., in a newspaper).

More specifically, let y_0 be the time-stamp of the previous round, and x_i the fingerprint for the i-th participant's document in this round, for $1 \leq i \leq m$, where m is the number of participants in this round. Then the time stamp for this round would be calculated as y_m, where $y_i = hash(y_{i-1}, x_i)$. The order of the x's is taken from the order of their owners' identifiers and is known to everyone.

A participant would calculate y_m for a round independently, from the broadcast values of all document fingerprints x_i, and from the time stamp of the previous round y_0. After the end of the calculation, to make sure he can deal with a challenge of his document's time stamp, every participant would maintain the fingerprint of all other documents in the same round. If a participant did not submit any documents in a round, he needs to remember only the round time stamp. Overall, a participant needs to remember one time-stamp per round, and a number of fingerprints dependent on the number of documents he has submitted and the number of overall participants.

The authors also describe a method to reduce the number of fingerprints remembered by every participant to a constant multiple of the documents he submits. This is done through the use of associative collision-free hash functions. Associative hash functions allow the order of hashing operations to be varied, as long as the order of the fingerprints remains the same. So, every participant just needs to remember the chained hash value of the fingerprints up to his document, his own fingerprint and the cumulative hash of the fingerprints after his own. Using the notation above, the owner of the i-th document would only need to remember y_{i-1}, x_i and $z_{i+1} = hash(...hash(hash(x_{i+1}, x_{i+2}), x_{i+3})..., x_m)$. Since the order of the hash operations doesn't matter, the round time stamp can be verified as $y_m = hash(hash(y_{i-1}, x_i), z_{i+1})$. This is a significant storage improvement for any practical implementations of this, since the number of participants per round is expected to be rather high.

15.2 A hierarchical scheme

In the absence of a convenient associative collision-free hash function, an alternate scheme can be used to limit the number of fingerprints remembered by each participant to a sublinear function of the number of participants, per submitted document.

Based on the order of the participants' identities, a complete sorted K-ary tree, whose leaves are all participants is known by everyone. At every round, every participant hashes his fingerprint with the previous round's timestamp and broadcasts it. Those not submitting documents in that round are assumed to have broadcast a default value. Once each participant has received all fingerprint values, it calculates the value of each node in the tree, hashing together the values of the node's children. The value of the root of the tree is the time stamp for the round. Each participant need only remember the values for all nodes in the tree that are siblings to his ancestor nodes. Using those values, the participant can always demonstrate, given his document's fingerprint and the sibling node values, how the hash function can yield the well-known round time-stamp.

This scheme, using a k-ary tree, requires that each of the m participants knows all time-stamps, and $(k-1) \times log_k m$ tree values for each round in which he submitted documents.

16 The Multics Virtual Memory: Concepts and Design (A. Bensoussan, C. T. Clingen and R. C. Daley)

[20]

This paper is mainly used as an indication of how difficult things would be if we still had physical memory limitations. The creators of MULTICS went to amazing lengths to make sure that no physical memory is wasted on stuff that's not immediately needed.

Some VM facts:

- There is a different page table per segment. A segment can have up to 64 pages, and since each page is 1024 words long, each segment can be up to 64K words long.
- A portion of physical memory is reserved for page tables only.

- The "i-nodes" of segments on the filesystem need to be modified whenever a segment is activated in the system (the address of the page table assigned to the segment is recorded in it). Doesn't sound too smart, although it might be justified by the constraints of the time frame. Recovering from a crash must be a lot of fun, with active switches left on all over the place in the file system.
- The number of page tables (i.e., "(PT,ASTE) pairs") substantially exceeds the number of pageable blocks of core. Funky! More pagetables than pages. Those were strange times...

Terminology:

- *Make Segment Known* It means associating a segment pathname with a segment number. If some other segment has made a target segment known, this task avoids "reacquainting" the process with the same segment. Intra-process scope.
- *Activate Segment* It means allocating a page table for a segment and copying the segment map into the active segment table. The segment on secondary storage is marked to point to the associated active segment table entry in physical memory. Extra-process scope.
- *Connect Segment* It means connecting a segment's descriptor to an active page table. In other words, if the segment is already in memory, brought by some other process, connection finds it and uses it. Interprocess scope.

17 Lightweight Remote Procedure Call (Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy

[23]

RPC is cool as a message passing mechanism. It's designed to work cross-machine. However, the most common case is still single-machine calls (as shown in studies of V, taos, sunos/nfs). The authors attempt to create an optimized version of RPC.

Locally RPC overheads come from:

stubs They need to be general, but are not often used fully.

copies across domains

access authorization

queuing

context switches

command parsing in the callee's dispatcher

Optimizations have been proposed but are not far enough.

LRPC calls go through the kernel, which validates the authorization and creates the link to the callee.

Clerks deal with interface exports. Interfaces are exported in structures much resembling virtual method tables. There's one in every domain. It registers with a name server. Procedures in an interface are given a descriptor, which contains an entry pointer in the server's domain and a list of argument stacks. A-stacks are mapped read/write in both domains, so no copying is needed to transfer arguments from caller to callee. A-stacks can even be shared among procedures of the same interface. Linkage records contain the return domain and address for an A-stack. The whole thing is contained in a binding object, which is unforgeable, much like a capability.

E-stacks are managed by the server. They are associated with an A-stack until the server runs out and reclaims one. E-stacks and A-stacks adhere to the Module procedure calling conventions, which separate argument from execution stacks.

More complicated stubs are done in a more conventional RPC way.

On multiprocessors, domains are cached on idle processors, so that redundant context switches are avoided.

In general, arguments are copied around fewer times. All kernel copies are eliminated. An extra copy is done when parameter immutability needs to be ensured.

Multiprocessor speedup is almost perfect, because locking of shared data structures is minimized.

18 Practical Considerations for Non-Blocking Concurrent Objects (Brian N. Bershad)

[22]

Main target: shared-memory multiprocessors.

Consistency can be guaranteed by mutex critical sections or non-blocking concurrent objects. Non-blocking objects have no (actually, only short) queues, i.e. are not vulnerable to convoys, priority inversion, deadlocks.

Basic problems:

- compare-and-swap(CAS) doesn't always exist
- performance in the presence of contention

Synthesizing CAS using locks, when no hardware CAS exists, is not non-blocking. *Roll-out* solves that, by always releasing the lock on preemption. The code resumes at a different point, depending on where it was when it was preempted. This mechanism seems to be there only for the CAS mock-up. Roll-forward finishes up on preemption and it's harder to implement (for instance, what do you do on page faults?). Software CAS is good for more complex conditions and multi-byte swaps. Roll-out can be prevented from progressing and is vulnerable to processor failure.

How do we synchronize to mock-up a CAS? Synchronization operations are becoming costlier in faster architectures.

Spin-locks cause a lot of bus activity on all processors: when the lock is released, everyone tries to acquire it.

Queue-locks turn lock contention into a relay race. Each contender waits for a single holder to release the lock. Therefore, every sync operation is a successful operation. They are more complex to implement and geared towards lock-based mechanisms. However, "success" here means that

1. the lock was acquired, and
2. that the CAS succeeded

This is not the case in non-blocking synchronization. Unfortunately, for a sequence of threads trying to accomplish a similar update on a shared data item, only one will succeed, and all else will fail, after they've waited to acquire the lock for the CAS mock-up.

A general fix is to only request a lock when the CAS is likely to succeed, by checking the CAS value, before acquire-lock (something like "compare*-compare-and-swap").

19 Extensibility, Safety and Performance in the SPIN Operating System (Bershad, Brian N. and Savage, Stefan and Pardyak, Przemysław and Sirer, Emin Gün and Fiuczynski, Marc E. and Becker, David and Chamgers, Craig and Eggers, Susan)

[24]

Extend the operating system services, to make the system correspond better to the requirements of specific applications. In this way, different clients can get independent specializations.

19.0.1 Motivation and Techniques

Provide

- extensibility (through interfaces and fine-grained access)
- safety (by avoiding exposure to actions of others, and controlling access at the same granularity as that of the extensions)
- performance (by minimizing the overhead between the extensions and the system)

All that is given through the following techniques:

co-location Put the code where it will be executed (in the kernel).

enforced modularity Enforce it through the type-safety of the associated language.

logical protected name spaces Don't export to the extension what it doesn't need to play with.

dynamic call binding/event handling Hook extensions on top of events inside the system.

Comparison to other approaches:

- Separation of mechanism from policy can be very complex for arbitrary systems.
- Microkernels export everything and let applications implement the services. The high communication overhead they induce leads designer to coarser-grain systems.
- Little languages try to limit how they can be abused by limiting their expressibility. However, that makes them less useful.
- Software fault isolation takes a brute force approach, since everything is checked at run-time. It doesn't contain or deal with the concept of an interface, i.e., a group of related functionalities.

19.0.2 Protection

Protection in SPIN is supported through capabilities and protection domains.

Capabilities are unforgeable language-level pointers. They're supported and protected by the language (Modula 3) and impose no run-time overhead. The kernel externalizes them through indices into a kernel-level capability pointer table (much like file descriptors in Unix).

Protection domains are interface name spaces. They're created from "safe" object files (i.e., object files signed by a trusted Modula-3 compiler, or asserted to be trusted by an administrator). Such namespaces can be combined, resolved and so on.

19.0.3 The Extension Model

The extension model is mainly based on events and event handlers. An event is any function call within the kernel and handlers are alternate or replacement implementations for these function calls. For example, a page fault trap is an event and a piece of code that handles such a trap is a handler for it. Handlers are linked to events by a central dispatcher. An event handler can be a procedure or a chain of procedures. Handlers can be applied selectively, if an associated predicate succeeds (these predicates are called *guards*).

19.0.4 Evaluation

The Dispatcher is a big bottleneck, especially when multiple guards need to be evaluated. Event handlers are, in general, very deeply trusted once their credentials have been validated. In other words, a trusted misbehaving extension (for example, a buggy extension), can wreak havoc in the system in many ways (an example is holding down a lock on a resource, causing starvation on threads of control needing that resource).

20 Bimodal Multicast (Ken Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky)

[25]

This paper starts with a very entertaining, epic description of the abstract problem, taken from the "world" of the Byzantine Generals. A general can win a battle if he can get a continuous stream of information to all of the troops, at regular times, with at most a given, calculated percentage of troops unreachable and only a calculated percentage of all messages in the stream undelivered. Behind all this, it is understood that stronger "reliability" guarantees, such as atomicity (if anyone gets a message, everyone gets it eventually), delivery ordering, virtual synchrony and so on may not be necessary in some settings, incurring great complexity and great restrictions on tolerated system conditions and scales. On the other hand, in the context of multicast, reliability has also been addressed as a means of best-efforts delivery (i.e., if a packet loss is detected, a reasonable — but potentially unsuccessful — effort is made to deliver it) to an otherwise anonymous, nebulous group of participants. Without controlled membership, the authors suggest, end-to-end reliability is an ill-defined concept. However, in relatively stable, low-loss networks, this definition of "reliable" multicast scales to much larger groups with reasonable, intuitive reliability. The trade-off is to choose between a scalable but nebulous notion of reliability and a strong notion of reliability that does not scale and can be sabotaged by frequent system instabilities.

This paper proposes a multicast protocol that scales well and provides *predictable* but probabilistic notions of reliability, even under significant network perturbations or system failures. The protocol does not deal at all with

Byzantine failures; it handles, instead, hard processor and link failures as well as soft failures (i.e., transient congestion, buffer overruns, etc.) Furthermore, it does not rely on synchronized clocks.

This is a multi-source multicast protocol, not a single-source protocol.

The protocol consists of an optimistic multicast phase, and an anti-entropy phase. In either case, group membership is considered constant and globally known. The authors say they use group membership mechanisms that scale up to a few hundred group members, but scaling up to thousands is the topic for further study.

The general idea is: perform IP multicast hoping for the best, then repeatedly tell random others what you've seen, ask others for what you discover you've missed. Phases are concurrent (i.e., there is no distinct switch from multicast to anti-entropy).

Message recovery via anti-entropy prioritizes the recovery of recent messages over older ones. This is in keeping with one of the stated goals of the paper, which is maintaining a steady throughput even at the expense of strict global consistency.

Messages are delivered to the application in FIFO sender order (though there may be multiple senders - no global ordering is built but it can be superimposed on top of pbcast). If a lost message cannot be recovered, an appropriate upcall is delivered to the application. Messages are garbage-collected a few rounds after they were received. Until then, they are gossiped about, in case someone else needs them.

During gossiping, a process chooses another at random, and sends it its message history. If during this exchange wholes are discovered that can be filled, the participants retransmit such messages to each other.

The authors extend the original protocol with several interesting optimizations:

- "Retransmission requests are only served if they are received in the same round for which the original solicitation was sent." This means that if a process is so slow as to attempt responding to a retransmission solicitation in a later time quantum than the one in which it received that solicitation, then that process ignores the request, since it is apparently experiencing troubles.

- Within each time quantum, there is a cap to how much bandwidth is devoted to message retransmissions. This prevents cascade failures caused by a recovering process that tries to catch up all at once.

- Processes cycle through their outstanding retransmission solicitations, to prevent redundant retransmissions. This would be the case if a prior retransmission is still in transit.
- Messages are retransmitted in LIFO order. This promotes system consistency at least in a recent time window. For time-sensitive messages, this also prevents maniacal lingering on long-lost, old messages.
- Logical time flows independently at different processes. The interpretation of logical time is an entirely local decision. The logical time of one process (i.e., its round number) is never meaningful to another, so synchronizing these logical clocks would be moot.
- The requirement for global membership knowledge can be mitigated by having a central membership server distribute random subsets of the global membership to individual processes.
- Multicast those retransmissions that seem to have been largely lost. This can also be scoped multicast, so that only a locality that uniformly missed a particular message gets it the second time around.

21 Grapevine: An Exercise in Distributed Computing (Andrew D. Birrell, Roy Levin, Roger M. Needham, Michael D. Schroeder)

[27]

Grapevine is a distributed messaging system, providing service for service location, naming and authentication. It contains an early presentation of a distributed email system. The system described bears close resemblence to how Sendmail and SMTP work.

A basic concept presented here is that of content independence. The transport mechanism, whether SMTP or the Grapevine transport protocol, doesn't concern itself with interpreting the content of messages. End-point applications deal with the interpretation of the payload of a message.

The level of security described in the paper is weak. A better version was in the works.

21.0.5 Design Goals

- Experimentation
- Overall service integrity cannot depend on correct client operation or on the correct operation of all servers. If a client at point A dies, another client at B shouldn't perceive any functional degradation. If a server A dies, then a client whose communication is not affected by server A should not perceive any functional degradation.
- If a message is accepted for delivery, then it will be delivered, or an error condition will be generated and returned.
- Decentralized administration should be allowed (for tasks like user admission, control of aliases, etc.)

21.0.6 User View

Users are registered in a database. User entries can be grouped within group entries. The entry for a user contains an authenticator, a list of mailbox locations and an address for synchronous communication (mainly used for server entries – not for people). A user's multiple inboxes are independent from each other and do not support the abstraction of single inbox; a user must check all of them for new messages.

Access control allowes an individual to be an owner of a group (in which case, he may arbitrarily change the entry at will), or a friend of a group (in which case, he may insert or remove himself from the group).

Only "signed" messages may be sent (i.e., the sender must authenticate himself to Grapevine, before he can send a message). Message polling is not authenticated. Successful message retrieval may cause messages to be erased from the protected inbox or retained until later.

21.0.7 Message Servers

Once a message is constructed, the message server authenticates the sender, flattens out the recipients list (i.e., recursively expands all groups contained therein) and validates it (returns error conditions for non-existent recipients). It then receives the message body and dismisses the user agent. The recipients are then split into "steering lists", i.e., lists of recipients per destination

message server. The message is shipped to each message server once, accompanied by the corresponding steering list. Transiently undeliverable messages are queued for later delivery. Race conditions (such as that occurring when a validated user is removed by the time the message is ready to be delivered) are dealt with using appropriate error messages, sent to the original sender. Messages are postmarked with the time at the first-hop message server. This postmark can be used as a hint for later sorting.

Messages are shared when delivered. When a message is addressed (and subsequently delivered) to multiple users with inboxes at the same message server, then the message body is stored on disk only once.

21.0.8 Registration Servers

The name space is split in two levels: the top level denotes the name registry and the bottom level denotes the person within that registry. Registries are distributed and replicated to multiple name servers. Replication is at the grain of a registry. Updates can be submitted to any server containing a registry. That server must propagate the update to the rest of the name servers serving the same registry.

Servers talk to eachother (for example, to propagate updates) using the same messaging protocol. So, a server with updated data will send a message to those server it knows also should hold that data. Servers check their messages every 30 seconds. Deleted items are retained along with their deletion timestamp, to deal with cases where an addition and a subsequent deletion arrive out of order. Every 14 days, deleted items are purged. Nightly, registration servers perform sanity checks by checking notes with other servers serving the same registries.

The metadata of the registration database are replicated everywhere. The metadata registry contains entries for all registration servers in service (as individuals) and for all registries (as groups of registration servers). To bootstrap the protocol with a registration server, two methods are used: an external naming service that resolves "GrapevineRServer" to a set of registration servers and a local broadcast service query.

Resource location works through the registry. Services have their own groups in some registry. Servers are members of the service they perform. A client retrieves the membership of a service and then picks that server which is best, according to some metric.

21.0.9 Caching

Caching is used to store hints about where an entry's preferred inbox lives. If a cache entry is outdated (for example, the preferred inbox has been removed), it returns a cache flush notification. No other explicit invalidation takes place.

21.0.10 Commentary

Some of the administrative functions (for instance, the membership function) sound inherently extremely expensive (e.g. determining group membership in a group closure).

A general characteristic of Grapevine is that it understands the semantics of the registered entries' structure (for example, it knows what a "connect site" is, in page 263). Marrying the registry with the transport module in such a way means that either one (or both) will have trouble evolving. For example, if the transport changes to a different protocol that uses more than one "connect sites", the registry API will have to change to accommodate the upgrade.

Authenticating users before messages are accepted implies that there's a trust base among multiple organizations: If organization A accepts a message from user B, does that mean that organization C should deliver the message to one of its users?

Assuming that a human will serialize modifications to a registry is a rather outdated idea. Global distributed registries cannot be assumed to be directly serialized by humans any more.

22 Implementing Remote Procedure Calls (Andrew D. Birrell and Bruce Jay Nelson)

[28]

Local procedure call semantics works well, so extending it to work remotely is the next step.

Before RPC, programming on a distributed environment was too complicated. The authors' major aim was to provide a more conducive programming environment for remote computation. Other aims were to make this environment adequately efficient, and reasonably secure.

Local procedure semantics is followed as closely as possible. Client and server stubs are created automatically from the interface descriptions.

A client binds to a particular interface (the type) and a particular instance (a machine and a process exporting that interface). The instance can be "any". The name server stores interfaces and their instances and returns the appropriate one when asked. Exported interfaces contain entry points of each procedure, a unique id for the instance and a call id, which identifies a request and its response, as well as their order.

Interfaces are capabilities. Access controls on the naming service exporting them transfers on to them.

The transport protocol is reliable (ACKs, timeouts, retransmissions, keepalives for long calls). Time based identifiers are used to distinguish responses from reset servers or duplicates. Exceptions are raised on network glitches. The protocol is tuned for short exchanges of small arguments. For larger exchanges it uses a bulk-transfer-like protocol, which is not as efficient as the state of the art bulk-protocol (to avoid complexity).

Exception handling is defined in the interface. Exceptions are passed back, in the place of return packets.

To optimize server response, idle server processes are standing by. If a server is expecting a call, it gets it. Otherwise, the call is dispatched to an idle server.

Lots of neat ideas that have been widely used since.

23 A Cryptographic File System for UNIX (Matt Blaze)

[29]

This paper presents a cryptographic file system, apparently the first proposed. The author motivates it by the inadequacy of both user-level, loose-application encryption approaches and of system-level approaches that live close to the hardware.

User-level cryptography that relies on external utilities, often used as filters, is cumbersome because it often results in "left-overs" such as unencrypted version of sensitive files, requires inconvenient per-application setup, and opens up windows of vulnerability as data flow between applications and the cryptographic utilities. Even when user-level cryptography is embedded

in applications themselves, it is frequently independently implemented per application, making interoperation of distinct applications on the same data difficult, opening up potential implementation weaknesses (since the same functionality is coded once per application), and leading again to windows of vulnerability, especially since some applications do not have the cryptographic facilities available to them.

On the other hand, system-level cryptography makes it hard to delineate the requirements of individual users. For example, secure disks make it hard to place different users' files (encrypted with different keys) on the same disk. At the system level, identifying which component is shared and which isn't is sometimes difficult for users who do not control those components. To protect files, a system must secure the physical media (both for current access and for backups), must secure the network when a remote file server is involved, and must even secure the file server itself. Often, building a secure network layer and the secure storage layer independently results in duplication of effort and grave inefficiencies.

The work presented in this paper, the Cryptographic File System, builds cryptographic facilities at the interface between the file system and the user, following the principle that data should be encrypted immediately before being sent to untrusted components, including potentially the file server itself.

The goals of CFS include rational key management (once per session per user, at the granularity of file system hierarchy), transparent access semantics (treating even encrypted files as normal files both below and above the user-kernel line), interactive-friendly performance that only impacts users who need the extra functionality, and protection of data and meta-data alike, both on the physical medium and in transit over a network.

Functionally, CFS operates by mounting (or "attaching") an unencrypted view of an encrypted file system directory in a special mount point in Unix. All operations on the unencrypted view proceed exactly as they would for unencrypted files. The cryptographic facility runs in the same machine as the user, even when files are stored remotely on separate file servers, removing the need to trust remote file servers for anything other than keeping opaque bits available.

Files are neither encrypted in the ECB mode of DES, since that could reveal the internal structure of files (e.g., that two blocks are identical). However, chaining modes of DES such as CBC are not appropriate either, since they require than whenever a block changes, all subsequent blocks must be re-encrypted, and that when a block is read, all previous blocks must be

decrypted first. Instead, CFS builds a long pseudo-one-time-pad for a given file using DES with chaining (OFB), which it XORs with file blocks before encrypting or after decrypting; the pad is retained while a file system is mounted and is discarded otherwise. To prevent comparisons of blocks at the same positions for different files encrypted with the same key, CFS uses different initial vectors for each file, which it can optionally save in an inode within the guid field.

The implementation of CFS is based on a user-level NFS loopback server.

24 Decentralized Trust Management (Matt Blaze, Joan Feigenbaum, Jack Lacy)

[30]

This paper describes Policy Maker, an early framework for certificate and certificate policy management.

It describes the main difference between web-of-trust services (such as SPKI/SDSI and PGP) and hierarchical or centralized certificate authority services (such as PKIX/X.509).

In web-of-trust services, I assign my own level of trust on each key I handle, and a separate level of trust on those keys, when they introduce other keys (key introduction is a certificate signed by someone, certifying that a key belongs to a certain third party). In PGP I can define how to use my levels of trust in deciding the trust I will place on introduced keys (e.g., if I have two distinct mid-level introductions, I'll assign mid-level trust on the introduced key). However, with PGP, I have no idea where to start, to locate the right certificate chain, if a communication from an unknown person shows up.

In centralized services, all Certificate Authorities (CAs) are linked together, and share a common trusted ancestor. So if I'm trying to find a key for someone, the CA tree is going to derive a chain of certificates that gives me a trustworthy answer. Having this nice happy family of mutually trusting CAs, however, is a bit of a pipe dream in a competitive environment.

Policy Maker defines a framework that, given policy and specific input certificates, can apply the mechanism and produce answers. It also ignores naming, and uses key/privilege mappings only, where traditionally, a key is

mapped to an identity (a name), and then privileges are indexed by identity.

Policy Maker uses two basic constructs: requests and assertions. Requests are used, when an application tries to get a policy answer on a particular set of inputs. Requests follow the format ‘‘<list of keys> REQUESTS <action>’’. The list of keys is the assortment of keys the application is able to use for a particular action. The action is an application-specific definition of an authority, such as “Name: Petros” for an identity action. An example of a request would be ‘‘<Petros>’ key REQUESTS N̈ame: Petros Maniatis¨’’. This request is trying to establish whether the key ‘‘<Petros’ key>’’ is allowed to assume the identity “Petros Maniatis”.

Assertions are the constructs populating the Policy Maker's database with facts about the local policies and transient certificates. Assertions take the form ‘‘source ASSERTS target WHERE filter’’. Source is either local policy, which is totally trusted (since the local application has defined it), or a certificate signer key. Target is either a key, upon which an assertion is bestowed, or a more complex construct, such as a threshold, n-of-k kind of target. Filter is a predicate which must evaluate to `true`, for the assertion to hold. For example ‘‘POLICY ASSERTS <Petros’ key> WHERE PREDICATE=regular_expression:Äbout: Petros¨’’ is a policy specifying that we can trust any certificate signed by Petros' key, if the target of the certificate is Petros himself. A certificate would be ‘‘<Petros’ key> ASSERTS <Petros’ message key> WHERE <time before Tuesday>’’. This is a unilateral delegation of authority from Petros' key to his message key, which will expire on Tuesday.

Filters are written in a safe language. The authors have devised a safe version of AWK.

25 Space/Time Trade-offs in Hash Coding with Allowable Errors (Burton H. Bloom)

[31]

This seminal paper introduces the concept of hashing with allowable errors. The target application is one that wishes to classify random arriving data elements as belonging to a small subset of the element space, when the subset is not otherwise delineable (i.e., cannot be expressed as a regular expression, for example). If a bounded number of false positives can be tol-

erated, then the problem can be solved with this lossy hashing scheme that later came to be called a *Bloom filter*.

The reason for tolerating some loss, when conventional hashing can do the job without losses is purely related to resources: memory resources for maintaining the entire hash table resident during lookups and time resources for accessing a potentially swapped hash table in the critical path.

The author compares three hashing schemes:

- Conventional loss-less hashing with linear probing. All elements in the subset are inserted into a large enough hash table. Random arriving elements are subsequently tested for membership in the subset via a straightforward lookup in the hash table.

- Lossy hashing with linear probing. All elements are first hashed so as to be mapped onto a smaller key space (with some loss). Then, the resulting digests are inserted into a regular hash table with linear probing. During normal operation, a new element is first digested, and the digest sought in the hash table. If the digest is found, then the element is said to match the subset.

- *Bloom filters*[1]. The hash table consists of a 2^n-element bit array. Every element in the subset is hashed into d different n-bit values using d different hash functions, and the bits in the hash table corresponding to each of the d different hashes is set to 1. During normal operation, a new element is hashed using the d hash functions. If all corresponding bits in the hash table are set, then the element is said to match the subset.

The author analyses the three hashing schemes based on how much space they require (the size of the hash table) and how much time on average they take to reject an element outside the subset, which is the usual case for the target application setting. Specifically for the two lossy methods, he proceeds to compare them for a fixed probability of allowable false positives. Note that the two techniques above never give false negatives (i.e., reject an element that actually belongs in the subset).

Although the derivation of the analyses is certainly readable, the salient point is that the third method, Bloom filters, has consistently better properties in the Space/Time tradeoff than the second method for the same prob-

[1]The author does not call them "Bloom filters" in this paper

ability of false positives. One key difference between the two, however, is that the second method can scale all the way up to become a lossless method (by replacing the digest function with the identity function), whereas Bloom filters have a maximum scaling factor (the number of hash functions used) where the additional benefit is outweighed by the additional cost (diminishing returns). This point is specific to the particular subset to which Bloom filters are applied: when the number of 1 bits in the hash table is roughly half of all bits, then a random element is as likely as a subset element to be considered positive.

26 A performance comparison of multi-hop wireless ad hoc network routing protocols (Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu and Jorjeta Jetcheva)

[32]

This paper compares four leading ad hoc routing protocols in a controlled scenario that isn't necessarily realistic, trying to draw conclusions on packet delivery ratio, routing overhead, and path optimality. Another very important contribution of this paper is its extension of the NS discrete event simulator to deal with ad hoc wireless networks.

The authors have extended the model of wireless communications present in the simulator, to deal with signal power attenuation, the intricacies of the medium access control protocols used as defined within the IEEE 802.11 standard, and the address resolution protocol.

Four ad hoc network routing protocols were studied: DSDV, TORA, DSR, and AODV. The authors remained relatively loyal to the specifications of the different protocols, although they did attempt to improve the protocols in some ways, such as their interactions with the address resolution protocol, and their interactions with the medium access control layer.

Everyone of the four different protocols had a score of variables that can be tweaked one way or another, that have to do with retransmission timers, buffer sizes and such. In most cases the authors take values for those variables that helped the convergence of the different protocols and lead to clearer results. It is unclear how disruptive those particular choices where.

The authors were not attempting to measure protocol performance on a particular workload taken from real life. Instead, they wanted to measure protocol performance under a range of controlled conditions. They simulated 50 wireless nodes forming an ad hoc network and moving about in a rectangular area of 1,500 m by 300 m for 900 seconds of simulated time. The mobility patterns of the mobile nodes were varied on a few axes. The overall characteristic of the mobility patterns used was called "random way point " model. This means that every node pauses for a number of seconds, then it takes a random destination in the rectangle and moves in order to reach that destination at a constant speed picked uniformly between a minimum and maximum speed. The duration of the pause between moves was varied from zero to 900 seconds. The maximum speed also varied between two values, 1 m per second and 20 m per second. The communication model basically consisted of a number of constant bit rate sources. The number of sources varied among three distinct values, 10, 20, and 30 sources. It's not clear how the destination of each source was picked. A steady rate of four packets per second was created from each one of those sources, and every packet had 64 bytes of content. The combinations of all these variables yielded over 400 different scenarios explored.

The mobility pattern used in the scenarios had the interesting effect on the lengths of routes over which each packet traveled: packets were uniformly distributed among routes with a length of one to eight hops. This appears to have happened after the fact, i.e. the effect measured was not intentional.

The first metric measured was *packet delivery ratio*. That was the ratio of the number of packets originating at the application layer sources over the number of packets received by the sink at the final destination. In that metric the two on demand routing protocols, DSR and AODV performed much better than the other two. In fact, they delivered more than 95 percent of all packets. DSDV performed rather poorly at the high mobility scenarios: rapid mobility changes are not propagated as efficiently throughout the network in the pro-active routing protocol, and while the routing fabric is stabilizing no alternative routes are available. TORA does relatively well although at high data rates it doesn't seem to be able to converge, chiefly because where congestion is present many packets tend to get lost between routing transitions.

The second metric used was *routing overhead*. That was that total number of routing packets transmitted during the simulation. For packets sent over multiple hops, each transmission of the packet at every hop counts as one

transmission. Lower layer overheads such as those produced by the medium access control layer, the address resolution protocol or the data link layer were not included, since they were specific to the current implementation setup and would be different for different selections of a data link layer. The basic difference among the four protocols is that for the pro-active representative, DSDV, the overhead is constant as mobility decreases, whereas the rest decrease their overhead as mobility decreases. DSR is the protocol with the least overhead in packets. This is due to the fact that DSR does a good job exploiting existing state distributed around the network. For example, before a route request is flooded throughout the network, all neighbors of the node are consulted in cases they have the route sought. In most cases, where the node that requests a route is the one exhibiting higher mobility than its neighbors, this single hop query can yield a route. AODV on the other hand, always resorted to fully propagating route requests, which resulted in about five times as many routing packets as those created by DSR. Furthermore, in the version of DSR simulated, nodes could eavesdrop on other nodes' route requests and corresponding replies. Although this could affect power usage at the eavesdropping nodes, since power usage was not a factor in this study, eavesdropping gave DSR an edge. TORA underwent congestive collapse at higher bit rates, and still used quite a few packets even in the no mobility case.

The last metric measured by the authors was *path optimality*. This was the difference between the number of hops the packet took to reach its destination and the length of the shortest path that physically existed through the network when the packet was originated. In DSDV and DSR routes were very close to optimal. TORA and AODV had a significant tail taking up to four or more hops longer than optimal for some packets. Note that TORA was not designed to find shortest paths.

One interesting observation is that though DSR appears to be the best overall routing protocol, if looked at from the point of view of overhead bytes as opposed to packets, it is inferior to AODV, since DSR is a source routed protocol, which means it has to piggyback the entire source the packet has to travel within the packet header. On the other hand, since the packets used were 64 bytes, the size of DSR's source route overhead was far more damaging than it would have been for larger packets. Perhaps a larger packet size would have done DSR more justice on this account.

27 Disco: Running Commodity Operating Systems on Scalable Multiprocessors (Edouard Bugnion, Scott Devine, Mendel Rosenblum)

[33]

The return of the Virtual Machines.

The goal is to take current operating systems to shared memory multiprocessors. Also, allow fixed-parallelism operating systems (like NT) to share more resources.

Main technique: virtualize everything (CPUs, memory, I/O devices). The virtualization layer is much smaller than a complete operating system, so it's also faster to develop and less prone to bugs. Fault containment comes naturally since each virtual machine can fail independently of others running alongside it (although more is needed for hardware fault containment, e.g., cellular DISCO).

The traditional problems with virtual machines have been:

- Overhead: Real CPUs have to be emulated, privileged instructions and devices virtualized and so on. Each copy of the operating system has to reside separately in every virtual machine's memory, and so do separate buffer caches. Similarly, multiple filesystems have to eaxist for each virtual machine.

- Resource management. How do you multiplex a single physical device among multiple entities that consider it exclusively theirs? Decisions have to be made without higher-level information from the OSes (is the CPU doing something useful, or just idle-spinning?).

- Sharing, communication. How do you take advantage that all these different systems run on the same machine to increase functionality?

Each virtual CPU is treated like a process in a conventional operating system. A table is maintained of most recent register contents (including privileged registers), TLB entries, so that virtual CPUs can resume where they left off, when scheduled on a real CPU. Virtual kernels are executed in *supervisor mode*, which is not quite as powerful as *kernel mode* in the MIPS architecture. When a privileged instruction is attempted by the virtual kernel, the trap is handled by Disco which changes the appropriate virtual privileged registers and jumps to the trap hanlder of the virtual kernel.

There are three levels of memory in DISCO:

- virtual (within a virtual machine - each process has memory starting at 0, but address 0 is mapped to a non-zero phyhsical address)
- physical (within a virtual CPU - each virtual CPU has memory starting at 0, but address 0 is mapped to a non-zero machine address)
- machine (honest-to-goodness physical memory)

When a virtual TLB is reloaded, we replace the mapping from virtual memory to physical memory with a mapping from virtual to machine memory. Any subsequent accesses to virtual memory get correctly translated to the corresponding machine memory without additional overhead. The "page table" that translates physical memory page numbers to machine page numbers is the *pmap*, which is keps for each virtual machine. The pmap also contains back pointers from physical memory page numbers to their corresponding virtual page numbers, to assist with invalidations when a page is take away from the virtual machine.

In MIPS, kernel code lives in an unmapped address space. Since this bypasses the TLB, it makes virtualization either impossible, or too slow (as would be the case if every single instruction had to be emulated to fix memory references). Therefore, OSes on MIPS have to be slightly modified to put their kernels in mapped memory (this is, in fact, a change on the Operating System ITSELF, however other architectures don't have such problems with memory mapping, so this shouldn't be a problem for Disco on them).

Virtual machines suffer more TLB misses than when the OS runs standalone. This is because the kernel also has to go through the TLB. Also, TLB misses are more expensive, since traps have to be emulated. To alleviate the problem, Disco maintains a second level, software TLB, practically extending the size of virtual TLBs. So, some virtual TLB misses are never sent to the reload code, since they're satisfied under the virtual machine.

Under the virtual machines, the NUMA-ness of the system is hidden from the operating systems. Pages migrate or get duplicated transparently to improve performance. Page access trends are measured through the cache miss counting facility provided by FLASH. When a page needs to migrate, the virtual TLB entries are invalidated and the page gets copied. Similarly, virtual TLB entries are downgraded to read share a page across multiple real processors. To find the right TLB entries, a *memmap* is maintained, mapping machine pages to sets of virtual machine/virtual address pointers.

I/O devices are virtualized through Disco-aware device drivers installed into the virtual OSes. Each device defines a *monitor call* command, which is used to pass all arguments to a device command to the monitor in one fell swoop. These calls perform virtual to machine memory translations for DMA requests, before a request is issued. Thereafter, the Disco device driver can talk to the device directly. Especially in the case of devices accessed by a single virtual machine, the I/O resource doesn't get virtualized at all, except for DMA requests. Disco's internal device driver interface much resembles that of other operating systems, to simplify the use of ready-made device drivers within it. Disco, in fact, internally uses IRIX device drivers.

Real, modifiable disks are not virtualized. Onle one virtual machine may mount a disk. Others can share data on those disks by using NFS, through the virtual machine that mounts the disk.

For non-persistently modifiable disks, the concept of a *copy-on-write* disk is used by Disco. This is for disks with code or read-only data, where modifications don't need to be kept after a virtual machine reboots. Those are read-shared under the virtual machines, so that disk accesses to their pages are almost instant, one the disk page is in memory. In other words, a secondary buffer cache exists under all virtual machines.

A network interfaces are virtualized and given their own data-link addresses on different virtual machines. When a distributed file system protocol like NFS is used, pages are shared instead of copied whenever possible.

Most commodity Operating Systems have some sort of base hardware abstraction layer, to simplify portability. Making minor changes on that layer, makes virtualization faster and easier. For instance, an OS can be made to use loads/stores to a specific address instead of certain privileged instructions that only read or change the contents of a privileged register.

It is extremely easy to use a specialized, thin operating systems that can scale up to the entire multiprocessor without much work. Such is the case for special operating systems supporting heavy scientific applications. Commodity operating systems can run alongside unburdened scientific applications, that will take up all available computing resources when they find them unused.

28 Time-Stamping with Binary Linking Schemes (Buldas, Laud, Lipmaa and Villemson)

[34]

This paper examines the concept of time stamping, as originally defined by Haber and Stornetta [77] and extended later [19]. It identifies efficiency problems with *relative temporal authentication*, an operation that produces a temporal ordering between two given time stamps. Specifically, in previous time stamping schemes, the cost of verifying relative temporal ordering of two time stamps and the size of the necessary verification data were linear in the "distance" between the two time stamps. Distance here means the number of time stamps that have been issued between the two time stamps in question. In this paper, the authors present a binary linking scheme that allows relative temporal authentication that requires only logarithmic verification cost and data size.

The binary linking scheme described here replaces the straight hash-chain used in previous time stamping protocols to link earlier time stamping rounds to later time stamping rounds.

The authors also argue that it is impossible to build an efficient (i.e., sub-linear cost) time stamping scheme that puts enough information in each individual time stamp to allow two time stamps to be directly comparable, without help from the time stamping service.

29 Optimally Efficient Accountable Time-Stamping (Buldas, Lipmaa and Schoenmakers)

[35]

In this paper the authors present a theory of authentication graphs, a generalization of the authenticated data structures pioneered by Merkle with hash trees [**?**]. By applying the authentication graph theory to a hash tree time stamping scheme originally proposed by Benaloh et al. [19], the authors put together a new authentication graph, the *threaded tree*. Threaded trees are better than the authors' previous antimonotone authentication graphs (called binary linking schemes [34]) in that they yield shorter authenticators (i.e., time stamps) without compromizing those authenticators' usefulness. The savings is due to the removal, in the new authentication graph, of re-

dundant information; where in binary linking schemes the intersection of two same-round time stamps was enough to prove the temporal ordering of the two time stamps, in threaded trees the intersection of the two time stamps has enough information to *generate* a proof of temporal ordering between the two time stamps.

30 A Logic of Authentication (Michael Burrows and Martín Abadi)

[36]

[36] This is an attempt at designing a logic for reasoning about authentication mechanisms.

The main constructs on which the logic is based are as follows:

- *Belief*: "P believes X" means that, to the best of its knowledge, principal P may act as though statement X is true.
- *Disclosure*: "P sees X" means that principal P has seen (or has been able to see) statement X.
- *Accountability*: "P said X" means that principal P has made (at some point in the near or distant past) statement X.
- *Jurisdiction*: "P controls X" means that principal P has the right to make statement X (for example, a secure time server has the right to make the statement "the time is now 12:08").
- *Freshness*: "fresh(E)" means that the expression E is believed to have been created recently (after a certain well-known event), and to not have been used before.
- *Authentication*: Principals P and Q are entitled and justified to use a shared key K ($P \overset{K}{\leftrightarrow} Q$); principal P is entitled and justified to use a public/private key pair K, K^{-1} ($\overset{K}{\mapsto} P$); principals P and Q are entitled and justified to use a shared secret X ($P \overset{X}{\rightleftharpoons} Q$).

- *Encryption*: Data X are encrypted with key K ($\{X\}_K$); data X are authenticated using secret Y ($\langle X \rangle_Y$).

Using these constructs and some common sence rules about authentication, a set of inference rules have been put together:

- *Message-meaning.* If P believes $Q \overset{K}{\leftrightarrow} P$ and P sees $\{X\}_K$, then P believes Q said X. Similar rules exist for public key encryption and for shared secret authentication.
- *Nonce verification.* If P believes fresh(X) and P believes that Q said X, then P believes that Q believes X.
- *Jurisdiction.* If P believes that Q controls X and P believes that Q believes X, then P believes X.
- *Key use.* If P believes $P \overset{K}{\leftrightarrow} Q$ and P sees $\{X\}_K$, then P sees X. Similar rules exist for public keys and shared secrets.
- *Superset Freshness.* If a subset of a set is fresh, so is the set itself. In other words, if P believes fresh (X) then P believes fresh (X, Y).

Some additional rules dealing with instantiation of variables and other inference tools are also included.

To use the logic described above to derive facts about authentication protocols, a transformation of those protocols to the vocabulary of the logic must first be performed. The transformation was arbitrary and interpretational in this paper, i.e., there was no formal method to perform it.

Given a protocol, once its idealized protocol counterpart is derived (i.e., its equivalent protocol in the vocabulary of the logic described above), the inference rules may be applied to it, to derive a concluding set of facts about the protocol: a set of opening assumptions are stated, which are augmented by inferred facts after each step of the idealized protocol. The goal of this process is to prove in the end that both parties of an authentication exchange have a common shared key for subsequent communication (i.e., A believes $A \overset{K}{\leftrightarrow} B$ and B believes $A \overset{K}{\leftrightarrow} B$). Some times, a stronger goal is necessary (or possible), namely that both parties know that the other party has the same

information about the exchange as they do, i.e., A believes that B believes $A \overset{K}{\leftrightarrow} B$ and similarly, B believes that A believes $A \overset{K}{\leftrightarrow} B$.

The authors proceed to apply their logical analysis to:

- The Kerberos protocol [160]. Freshness of nonces is dependent on synchronized clocks, which tends to be a bit of a problem in practical implementations (especially since secure time servers are not widespread).
- The Andrew secure RPC handshake. Here a weekness is discovered, where one of the participants may be prone to replay attacks.
- The Needham-Schroeder public key protocol [127]. The authors discover a weakness in the interaction of principals with the certification authority (they arbitrarily believe the freshness of messages coming from the CA).
- The X.509 protocol (version of 1988). Here both redundancies and weaknesses are discovered. The weaknesses stem from facts stated in the X.509 specification which contradict what seems to be assumed in the protocol (namely, an arbitrary freshness decision about one of the used nonces).

30.0.11 Issues

The constructs and the mechanics of the paper are pretty interesting, although lots of pieces are missing, most notably the transformation mechanism into idealized form.

31 Application-aware Admission Control and Scheduling in Web Servers (J. Carlström and R. Rom)

[38]

This paper presents an analytical framework for improving the performance of services, via on-line estimation of admission control and scheduling parameters. The authors model a service by allowing interactions to be classified into "stages"; in a web service context, stages might be "just came in,"

"browsing with empty shopping cart," "browsing with non-empty shopping cart," and "checking out." Different rewards are associated to the different stages and the framework seeks to maximize those rewards by allocating resources to each stage, under the assumption that more resources (i.e., better service) reduce the probability of the client aborting the service, removing all reward potential.

Given a maximum rate of admitting new sessions, and a set of weights for allocating resources to each stage, the model allows a server to estimate its future service rate, its utilization per stage, its probability of retaining customers per stage, and its transition probabilities from stage to stage. Using those, the server can figure out its estimate for a "rate of reward" (i.e., how many brownie points it accrues per unit time) as well as the mean delay for each stage. Using these computations, optimal allocation weights can be computed for given observed incoming request rates, ensuring maximum rate of reward.

32 Practical Byzantine Fault Tolerance (Castro, Miguel and Liskov, Barbara)

[40]

This paper describes a replicated file system designed to withstand byzantine failures. The file system is built on top of a fault-tolerant replication algorithm, which can be applied to other problems as well. The total number of replicas, as well as their identities and authentication tokens, are known in advance, and like all byzantine failure tolerant systems, no more than a third of all the replicas may be faulty, for the systems to remain usable.

The algorithm follows the state-machine replication model. It does not rely on synchrnous communications for safety. Therefore, denial of service attacks cannot cause the system to give out incorrect information. However, the algorithm does rely on synchrony for liveness, so DoS attacks *can* stop it from making progress temporarily. Privacy is not preserved in a fault-tolerant way.

At any time, each replica has a *view* of its fellow replicas that are currently non-faulty. In every view, one of the participating replicas is deterministicly chosen as the *primary*. This replica coordinates actions on the replicated system. When this replica is detected as faulty, the remaining replicas in the

view gang up against it, and change views so that the faulty primary is no longer included.

A client submits a request with the primary of the current view. The primary propagates this request to the other replicas in the view. Those execute the request and return a response to the client, directly. Once the client has received $f+1$ response from distinct replicas, all of which agree on the result (thereby verifying that at least one non-faulty replica shares that opinion on the result), it assumes it has received the correct result. However, the client might not receive enough answers (e.g., it missed a view change, and only evil faulty replicas responded). Then, the client can broadcast the request to all replicas (since all are known in advance), and the real primary among them is going to take care of it as already described.

The primary imposes a total ordering on client requests by assigning each request a number. All the replicas have to agree on the order of requests, before they start executing them. The resulting protocol has three phases: *pre-prepare*, *prepare* and *commit*.

The first two phases help ascertain the sequence numbers of requests within the same view, thereby enforcing a total order to requests within the same view. When the primary wants to submit a request to the rest of the replicas, it sends them a pre-prepare message. Replicas accept the pre-prepare message if it has a new sequence number, and belongs to the current view. Then, every replica that received a preprepare message sends a prepare message to everybody else. Once a replica which has already pre-prepared receives prepare messages from 2/3 of all the replicas, it is ready to commit, since it now knows that the ordering of requests it thinks is true is also shared by enough non-faulty replicas.

When it is ready to commit, a replica sends a commit message to all other replicas. After it has received matching commit messages from more than 2/3 of all replicas, and it has executed all requests with lower sequence numbers in the view, the replica executes the operation and sends a reply to the client. Note that although a replica may commit a request out of order, it still executes commited requests in order, based on the agreed-upon sequence number within the view.

For all the stages above to take place out of order, a fairly extensive message log has to be maintained at every replica. Periodically, every replica produces a checkpoint message on its state. This signed message tells all other replicas of the digest of the replica's state. Since all replicas must have the exact same state (the definition of state replication), if all replicas

take checkpoints at the same sequence number, then checkpoint messages should all match (as far as the state digest is concerned). Therefore, when a replica receives matching checkpoint messages from more than 2/3 of all replicas, it can be assured that it has a proof of correctness (or at least, agreement) on that checkpointed state. At that time, log messages predating the checkpointed state can be purged. To ensure forward out-of-order progress, without allowing message logs to extend indefinitely, replicas limit how many requests they can tolerate in their log, beyond the last stable checkpoint. If a request has a sequence number higher than an agreed-upon number beyond the sequence number of the latest stable checkpoint, it is ignored by a replica in the pre-prepare phase.

Everytime a replica decides it has been waiting too long for a particular request to execute, it initiates a view change, assuming that it's a faulty primary that's causing the stall. A *view-change* message contains:

- the intended new view number
- information about the last stable checkpoint known to the initiating replica, including a proof of its correctness (i.e., checkpoint messages from at least 2/3 of the replicas) and the associated sequence number
- information about all prepared but uncommitted requests known to the replica, including the pre-prepare message, as well as a proof of the correctness of the preparation of the request (i.e., prepare messages from at least 2/3 of the replicas)

When the primary of the new view receives view-change messages from at least 2/3 of the replicas, it decides it's time to enable this new view, by sending out a *new-view* message to all replicas. The new-view message contains a proof of the new view necessity (i.e., the view-change messages from at least 2/3 of the replicas), and a description of the initial state of the new view. This description contains a new pre-prepare message for every request prepared in the previous view (as evidenced by the view change requests), starting with the sequence number right after the latest stable checkpoint in all the view-change messages. Since different replicas requesting a view change might have different known stable checkpoints, the latest one among them is considered the right one.

If a replica finds out it is out of date, relatively to the advertized latest stable checkpoint, it asks one of the replicas certifying the checkpoint in the new-view message for a state dump.

To avoid excessive view changes, the time period after which a replica initiates a view change increases exponentially on subsequent expirations. So the limited synchrony, which as mentioned earlier is relied on for liveness, is used to regulate the rate of view changes. Given that no more than a third of the replicas is assumed to be faulty, the protocols will have to wait $2^{n/3}$ times the base timeout, if all faulty replicas happen to become primaries in sequence.

To improve performance, digital signing has been replaced by Message Authentication Codes (MACs) in all regular protocol exchanges, except in view changes. MACs are faster to produce, but bear no proof as to the author of the MAC'ed information. The authors explain how they had to modify the basic algorithm to deal with this difference. The resulting improvement justified the complexity.

Issues This was an interesting paper overall, since it presents something practical that deals with Byzantine failures. However, it still has a rather limited scope, since you need to know in advance the identities, keys and number of all replicas and faultiness is assumed to be static which makes the contraining assumptions rather vague. Besides, although promising, the performance results are rather incomplete, since no view changes are included in the experiments run. Perhaps in a cluster environment view changes need not be frequent, but then again, the whole argument of dealing with Byzantine failures isn't all that powerful in a cluster environment.

33 Secure routing for structured peer-to-peer overlay networks (Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron and Dan S. Wallach)

[39]

This paper introduces a secure routing primitive within the context of structured overlay networks. The secure routing primitive ensures that messages sent by a correct node to a key k are received by all correct nodes responsible for k. The authors propose to produce such a primitive by addressing three key problems: secure node identifier assignment, secure routing

table maintenance, and secure message forwarding.

Secure Node Identifier Assignment Secure node identifier assignment prevents an adversary from mounting attacks against the overlay that rely on defeating the randomness of node identifiers, which is key to the scalability and robustness characteristics of structured overlays. The objective of such an attack may be to place strategically nodes controlled by the adversary near a targeted resource or between two partitions of the entire overlay population. The adversary can then slow, alter, or drop critical messages trying to reach that targeted resource or to cross the barrier between partitions.

The authors recommend central assignment of node identifiers as the solution to the problem: a centrally trusted entity, much like a certification authority, issues node identifier certificates, observing the randomness requirements for the assigned node identifiers. Then participants in the overlay are not only prevented from attacking, but are also given the means to communicate securely with each other. Certificates bind not only a public key to an identifier, but also an IP address, to prevent adversaries from taking advantage of proximity optimizations based on the location of individual IP addresses; this defense, however, limits the flexibility of the solution, especially in the case of dynamically changing IP addresses, as for example with mobility. To limit the *rate* — as opposed to the distribution — of identifiers, the authors further suggest that node identifier certificates have an issuance cost, that can render expensive attacks involving the iteration through many certificates. Alternatively, certificates could be bound to real-world identities, which can be convenient within a single administrative domain.

This solution for the secure node identifier assignment is preferable to alternatives such as distributed identifier generation. The authors point out that Sybil attacks [61] can make such generation vulnerable to tampering. The authors mention as minor improvements solutions that involve cryptographic puzzles to rate limit identifier generation, but also a combination of cryptographic puzzles with decentralized identifier certification, based on public key cryptography. Specifically, a node can use a public key pair with the property that its public key hashes (using SHA-1, for example) to a value ending with p zero bits; p is the security parameter. A different hash of that public key can then be the node identifier. Such certified identifiers can also be massaged to bind to IP addresses. Finally, the authors suggest

"perishable" node identifiers, that require constant effort in reasserting one's identifier, causing however unnecessary costs on good nodes.

Secure Routing Table Maintenance Route table maintenance must be such that the probability of having an adversary-controlled entry in a correct node's table is the same as the probability that a node is controlled by the adversary. Secure node identifier assignment is not sufficient to obtain this goal; the authors offer two attacks demonstrating this:

- The adversary can exploit proximity measurements to cause correct nodes to place adversary-controlled nodes in the routing tables. For example, when a correct node contacts an adversary-controlled node to obtain latency information, the adversary can have another of its nodes—one that has lower latency to the correct node—respond, pretending to be the node that the correct one intended to measure. The more nodes the adversary controls over the entire overlay, the more opportunities he has to fool correct nodes. The authors suggest that ensuring that only the intended recipient of an IP message can observe that message is a possible solution; this is not entirely correct since an adversary with global instantaneous awareness can even defeat that solution by reducing the return latency of probes. The authors also point out that well-funded adversaries may in fact have control over intermediate ISP routers or other forms of indirection such as mobility forwarders, making this a productive attack.

- The adversary can also just lie! When asked for routing table entries by a correct node, the adversary can always respond with entries pointing to its own nodes. Because good nodes some time have adversary-controlled nodes in their routing table entries whereas adversary-controlled nodes always have fellow adversary nodes in their suggested routing entries, this attack is guaranteed to cause correct nodes' routing tables to be eventually full of adversary-controlled entries.

To combat such attacks, the authors suggest two remedies: the use of multiple bootstrap nodes and the use of a constrained routing table, alongside a regular Pastry-style routing table.

To obtain its leaf set, a newcomer node asks multiple bootstrap nodes for the appropriate leaf set, and then filters out the closest ones to its node

identifier. As long as at least one of the bootstrap nodes is not adversary-controlled, the new node will find a good leaf set.

To do actual routing beyond its own leaf set, the node uses the regular Pastry-style routing table; when however the is experiencing unwelcome characteristics such as high latency or high loss rate, it uses the constrained routing table, which is harder to subvert.

The constrained routing table mimics routing tables in Chord: a routing table entry at level l is the node one 2^l-th of the identifier space away, whereas in Pastry that same table entry may have the closest node (in some network metric) that shares l identifier bits. Although routing performance can be suboptimal using the constrained routing table, at least a node has a higher probability of avoiding adversarial behavior.

A node obtains its constrained routing table's entries by asking its leaf set for their entries. It then filters out the appropriate entries to insert into its own routing table. The assumption here is that, in the beginning, the node has no recourse but to trust its initial leaf set, and that this leaf set has all the necessary information to build its own routing table. A side effect is that, unlike with the Pastry-style routing table, this constrained routing table offers a "symmetric" view of the overlay; as a result, a node can notify nodes it places in its own constrained routing table entries to update their own entries with it.

Secure Message Forwarding Once the fraction and distribution of adversary-controlled nodes across the identifier space, and over correct nodes' constrained routing tables have been controlled as described above, the next step is to ensure that message forwarding fulfills the secure routing primitive. Alone these first two tools are not sufficient.

Routing to a single root key may fail when any of the hops from the sender to the root key is controlled by the adversary. For an h-hop path, and assuming that the first two tools ensure that each hop has a probability f of being controlled by the adversary, the probability that the path will successfully reach the root key is $(1-f)^{h-1}$. Taking under consideration that the root key may be handled by an adversary-controlled node, the probability that a correct root-key owner receives a message becomes $(1-f)^h$. By making a root key handled by a replica set (i.e., the leaf-set of the node that is closest to the root key), overlays seek to increase the chances that a correct node handles a particular request for a root key. To increase the probability above

give f, an overlay could decrease h, the number of hops between any nodes over the overlay. Although this can happen by increasing the number of neighbors per node, it also increases the cost of maintenance; in fact, for reasonably high levels of routing assurance, impractical numbers of per-node neighbors are required.

The authors approach this problem by instituting the concept of a *failure test* for routing. When the routing of a particular request to a key tests "positive" using the fast routing method, then the slower but more secure routing method over the constrained routing table and with redundant routing (see below) are employed.

Once a request is sent to a root key, the sender expects a confirmation from each of the nodes in the key's replica set. If not enough nodes respond, the test is assumed to have yielded a positive. When enough responses arrive, the sender evaluates the node identifiers of the responders to make a guess on whether those identifiers are likely to be all controlled by the adversary or not. The test relies on the premise that correctly assigned node identifiers are uniformly distributed over the identifier space; therefore, one would expect that the "density" of consecutive identifiers is roughly equal at any portion of the identifier ring. In fact, the test compares the average numeric difference between the consecutive identifiers in the sender's leaf set to the average numeric difference between the identifiers in the responding set of root key replicas. If the latter differs more than a factor γ from the former, then the failure test yields a positive. Note also that unverifiable node identifier certificates in the responding set, malformed replica sets (i.e., not centered around the root key in question), etc. also yield positives.

Unfortunately, the technique can yield false positives (when a correct replica set happens to have too great an average identifier distance) or false negatives (when an adversary-ridden replica set manages to have a low enough average identifier distance). From analytical estimates of upper bounds for the probabilities of false negatives and positives, the authors conclude that more "samples" for the computation of these averages can yield lower false answer probabilities. Extending the size of leafsets at the sender can help, but becomes harder for the receiver.

The authors moderate their hopes for the usefulness of the failure test by pointing out that the adversary can attack the mechanism fairly easily by *suppressing node identifiers*: the adversary can artificially increase the average identifier distance at the sender by removing nodes it controls around the sender from the overlay, thereby extending the breadth of the leaf set; and

by keeping a few correct node identifiers from the receiver's leaf set in the responding set of replica nodes, which decreases the average identifier distance from a corresponding fully-controlled replica set. Furthermore, the adversary can occasionally also suppress node identifiers in the root key replica set, which can throw off the sender's calculations. Such attacks can be mitigated with more communication, that aims to establish that what the sender is told about the vicinity of the sought root key is consistent with what the correct nodes in that vicinity think. Also, the sender can tolerate a higher probability of false positives in favor of a lower probability of false negatives; false positives yield to lower performance, but are compatible with the goal of the secure routing primitive.

When a positive occurs, the sender uses *redundant routing* over the constrained routing table. Specifically, it forwards r copies of the request to its leaf set nodes, all of which then forward the request to any node that contains the root key in its leaf set. Such nodes can then use local information to ensure all replica nodes receive the request. The process starts by discovering in this method a set of likely neighbors of the root key, who respond (in signed, fresh messages) with their local view of the root key vicinity. The sender then puts together a likely replica set, which it sends back to all the members thereof. If such a member thinks a node is missing from this replica set, it responds with its changes and, repeatedly, the sender attempts to converge to an agreed upon replica set. This technique, although expensive, has a probability of successfully reaching all correct replica nodes that is greater then 99.9% for $f < 0.3$.

An approach not taken by the authors is that of *checked iterative routing*, i.e., having the sender make per-hop decisions on where to send the request next. The technique doubles the latency of communication but does not seem to have a commensurate increase in success assurance. The authors speculate this is because iterative routes are less independent than recursive routes. They do not seem to justify this conjecture.

All of the techniques above can be significantly assisted by the use of self-certifying data, i.e., data that the client can check for correctness. Although this cannot be used for discovery of unknown data, or for publication of new data, it can be very helpful for the common case in which known-to-exist data re sought. In such cases, secure forwarding is only necessary when the efficient mechanism yields data that are invalid or ends with a timeout.

34 Designing Directories in Distributed Systems: A Systematic Framework (Chandy, K. Mani and Schooler, Eve M.)

[41]

This work seems to be attacking directories from a knowledge-theoretic point of view. The idea is to use directories in a distributed system to share information about processes among processes, given constraints on latency and overhead. When constraints limit the amount of knowledge sharing, then knowledge estimation is used.

The concept of process knowledge, as defined in the theoretical literature, is too strong to cover practical distributed systems. The problem is that "knowledge" is too strong a word. Traditionally, knowledge of a fact X about process B by process A means that fact X is true about process B. Unless something is done by process A to change its X-knowing state, fact X will have to remain true regardless of process B's will. Since in practical systems the state of process B can change with or without A's consent, the concept of knowledge cannot be supported.

An argument is made about the dependency between global clocks and knowledge gain or loss. The argument is connecting the fact that if I know X is true between times a and b, then at time a I gain the knowledge that x is true, whereas at time b I lose that knowledge. However, that seems a bit naive, since the knowledge was "x is true between times a and b" all along; the knowledge is not lost after it has become non-applicable. A minor philosophical point.

Estimation is based on conditional probabilities. In practical systems, a process can know a fact about another process only with a probability lower than 1 usually. This work tries to derive useful conclusions from sets of such probabilities connecting different facts in a distributed system.The goal for a designed directory would be to optimize the estimate probabilities for the contained properties, for a given set of infrastructure constraints. Problem-specific properties, such as temporal or spatial locality can affect how these estimate probabilities vary. Moreover, ways to increase probabilities are explored, as well as the price incurred.

The directory design space is broken into two axes: organization (centralized, hierarchical, distributed) and maintenance (active or passive, push or pull).

- centralized directory: this is good when the sources of information live near the directory, or the overhead per source is low, or updates are relatively infrequent. Scaling and reliability are two major concerns about centralized directories.

- hierarchical directory: this is good for cases where common directory lookups are limited within a restricted-scope sub directory. In those cases, answers of higher quality (higher estimate probability) can be requested from higher levels of the hierarchy, which may be more expensive to access.

- distributed directory: this provides the greatest immediacy of state information, provided that applications permit inconsistencies among directories or require only eventual consistency. Distributed directories can only be viable when data replication is relatively inexpensive and the aggregate overhead in communication does not overwhelm the infrastructure.

- passive or periodic update (push or pull): prearranged updates are good when the period of validity of the transferred information is known in advance or when the information consumer needs periodic affirmation of this validity. The actual time period need not be static; it may adapt to current conditions or needs. However, the tradeoff to consider is between empty (i.e., redundant) updates and the cost of update requests.

- active push: this happens when the reporting process lets the directory know of a significant state change. Since updates are pushed to the directory, and perhaps subsequently pushed out to subscribers to those updates, only useful update messages are propagated. However,if an update is lost due to infrastructure unreliability, it may be completely missed by those interested until the next significant update comes up.

- active pull: this makes sense when there are far more information consumers with far more requests than information producers and state changes. In those cases, the directory belongs near the requesters and the producers may be far away.

The paper proceeds to describe the simple case of a two-state process communicating with a directory, in both lossless and lossy communications

environments. Finally a specific case study is presented for a videoconferencing application.

35 Hive: Faults Containment for Shared-Memory Multiprocessors (Chapin, John and Rosenblum, Mendel and Devine, Scott and Lahiri, Tirthankar and Teodosiu, Dan and Gupta, Anoop)

[42]

It is an operating system for a shared-memory multiprocessor. It strives to minimize the impact of failures of individual processors to the operational state of the entire multiprocessor. Main tools:

- There is an independent kernel (called cell) running on each processor. Only the cell where a fault occurred crashes.
- Very few kernel resources are shared by processes running on different processors. Scalability can therefore increase better as processors are added.

This approach creates the following challenges:

- Faulting processors must be prevented from corrupting the rest.
- Resources must be shared, despite fault containment.
- The cellular set of kernels must present a single system interface to the outside world.

Fault containment has three components:

- Protection from identifiably misguided processors. Pages can be safeguarded from wild writes by *firewalls*.
- Recoverability from identifiably faulty modifications. Pages written by faulting processors are preemptively discarded.

- Detection of misguided processors. At first, it's enabled by heuristics, then confirmed by a consensus protocol.

A firewall is a feature of the FLASH architecture, whereby a processor can associate a 64-bit vector which each memory page in its local memory. Each bit in this vector represents a processor in the system (if there are more processors than bits, some sort of aggregation is done). A write attempt by a processor not allowed to write a remote page results in a bus error.

Memory sharing can occur in two ways:

1. Active sharing of a page used by processes on multiple cells
2. Loans of pages to cells with a shortage by others.

Resource allocation is handled by *Wax*, a user-level process running on all cells. Cells sanity-check input received from Wax; therefore, faulty operation of Wax can only hurt performance, not correctness. When a cell is rebooted because of failure, Wax is terminated and restarted across all remaining cells, once the system is stable again. This simplifies its structure.

Faults can appear in both software and hardware. In hardware, the interconnect can malfunction because of a faulting processor, similarly, a faulting processor can cause the memory lines it has access to to misbehave. Similarly in software, the errant kernel is the component responsible for fault containment. Therefore, when the kernel is misbehaving, wild writes cannot be avoided, unless there's hardware support (for example in the coherence controller), or a trusted, lower-level software component (as is the case in a microkernel architecture).

Three types of fault "contamination" have been identified and dealt with:

Message exchanges All received results from RPC are sanity checked. Requests are assigned timeouts to avoid waiting indefinitely.

Remote reads Cells are careful not to be crashed while reading another cell's internal data structures (through bus errors), dangling pointers (by checking alignment and allocator tags), or unprotected data modifications (by copying structures locally before checking them).

Remote writes The FLASH firewall mechanism prevents cells from writing other cells' internal data structures. All user pages used only by local processes are protected from remote modification. Otherwise, all pages

that *could* have been touched by a remote cell pronounced faulty are discarded.

All cells monitor all other cells for signs of insanity using heuristics (for example, timed-out RPC calls, bus errors trying to access remote memory, remote clock monitoring, failed consistency checks on remote, internal pointers). Once an indication has been registered, normal operation of the system halts temporarily, and all cells engage in a distributed consensus protocol, to decide whether the suspect cell is in fact faulty. If the consensus is affirmative, then the faulty cell is rebooted. This second phase guards against insane cells pronouncing every other cell faulty and automatically rebooting them. Furthermore, cells accusing others of insanity, if voted down twice in a row for the same accusation, are considered corrupt.

Recovery is performed in two phases. In the first phase, TLBs are flushed, remote mappings are removed from process address spaces. After a first barrier has been crossed by all, the second phase cleans up virtual memory, by resetting all firewall write permissions and discarding pages that were writeable by the faulting node. After the second barrier is crossed, cells can resume operations.

36 Untraceable Electronic Mail, Return Addresses and Digital Pseudonyms (David Chaum)

[43]

This is the seminal work that introduced the concept of routing *mixes*. These are forwarding servers with one main goal: To break associations between their inputs and outputs. This is useful because it prevents eavesdroppers from tracing a message from its sender to its recipient.

Mixes safeguard against linking of their inputs to their outputs using three techniques:

- They encrypt messages, including their headers. The message body for the same message is never the same as it arrives and as it leaves a mix.
- They batch and reorder their outputs. The order in which a message was received at their input cannot be used to determine the correspondingoutput message.

- They pad and limit messages to maintain a uniform message size. Message sizes cannot be used to associate inputs and outputs.
- They keep track of recently processed messages, to avoid replay attacks. They only forget about processed messages after they change cryptographic keys.

Mixes can be *chained*, i.e., used one right after the other. The originator of a message picks a sequence of mixes and hides it within the message using encryption. Every mix in the chain knows only from which host it received a message (the originator or another mix) and to which host it is supposed to forward it (the destination or another mix). Otherwise, a mix doesn't know how many mixes are involved in the forwarding chain, or the identity of the other hosts involved in the transfer. Because of this property, a dishonest mix, or even a number of colluding dishonest mixes, cannot learn anything about the sender and receiver of a message, as long as they do not occupy the beginning or the end of a mix chain.

In the preceding discussion, only one-way communication is assumed. Furthermore, the sender can remain anonymous to the recipient of communication. The author continues to describe a mechanism to embed an untraceable return address along with a message. This allows the recipient of the communication to reply to its sender, without knowing who that sender is. The embedded return address contains a return path back to the untraceable recipient, as well as a set of encryption keys to be used with the return message.

Combined, the two mechanisms can constitute a general purpose mail system, where pseudonym mail-blocks (the embedded untraceable addresses described above) can be broadcast publicly. An individual can pick one (based on a pseudonym associated with the block) and use it to send a message to its owner. Both ends of the communication can be anonymous then.

37 The Rio File Cache: Surviving Operating System Crashes (Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, David Lowell)

[44]

The basic idea is to improve filesystem performance and reliability, by preserving memory across crashes, to achieve write-through reliability with write-back performance. Crashes, not power outages, are the main villain attacked.

Goal: Survive operating system crashes without writing data to disk. They eliminate all reliability induced disk writes, by protecting memory through a crash.

To protect the file buffer cache in kernel memory, they use TLB protection: the buffer cache is read-only, until the kernel explicitly wishes to change it. On CPUs that support direct use of physical memory, bypassing the TLB, either that feature is disabled if possible, or "sandboxing" is used, by instrumenting the kernel to only allow writes, if the accessed memory is outside the file cache or authorized to be written. This is much slower than just disabling TLB bypasses.

To restore the buffer cache, an in-memory *registry* is maintained and used, where all buffers get tagged with their defining metadata (physical memory address, device number and i-node number, file offset and size). The registry only imposes 40/8192 or 0.5% of the buffer size space overhead. Before the VM and FS are booted during a warm reboot, memory is dumped into the swap partition. Also at this point, metadata are restored on disk, making sure the filesystem is intact. When the system has completely booted, a user-level process analyzes the memory dump and restores the file cache.

Effects:

- No reliability-induced writes needed
- Metadata updates must be carefully ordered
- Atomic updates to the registry are feasible

To test reliability, they use synthetic, injected kernel faults: bit flips, instruction mashing, high-level scrambling (variable instantiation eliminations, overruns in bcopy, switches of loop conditions, futzing with loop base pointers, ...). To detect corruption, they checksum all buffers and go through them to look for inconsistencies after the crash.

Rio is neat: much faster than other means of protection (like turning all writes into write-throughs). More reliable than pure write through.

Catch: it requires that the CPU do not erase the memory across a reset!

38 Quantifying Network Denial of Service: A Location Service Case Study (Chen, Bargteil, Bindel, Katz, and Kubiatowicz)

[45]

This work looks at the problem of denial of service from a simulation standpoint.

The authors evaluate a centralized, a replicated, and a distributed directory service against distributed denial of service attacks that use flooding and corruption. The centralized service just responds with an answer to a lookup requst. The replicated service places a fixed number of copies of the centralized service on distinct servers, ignoring the consistency traffic. The distributed service uses a particular distributed hash table (Tapestry) to partition the lookup functionality over a large population of end hosts.

Normal (non-attack) workloads are modeled using a Zipf distribution, and a hot-cold distribution. Item sizes is uniform over an interval of relatively small sizes (5-50 KBytes). The flooding attack workload consists of fixed-rate bit streams sent from a varying number of sources to specific "important" nodes (e.g., the directory server, a server replica, etc.). For corruption, the authors look at two specific attacks: corruption of routing state so that the victim appears close to everyone, and corruption of hashing state (for the distributed example) where a compromised nodes appears to have authoritative information about all items, always responding with negative answers.

For flooding, first the authors compare the centralized service with a distributed service and conclude that the damage that a distributed attack can do is greater than what a single attacker can cause, even when the actual size of the flood traffic is the same between a single and a distributed

attacker. This is due to the limited bandwidth (bottleneck) available to a single attacker. Also, the distributed service is more resistant than the centralized service, primarily because of the load-balancing effect of caching results in the overlay (i.e., attacking the overlay node that's authoritative for a popular item does not limit the ability of other overlay nodes who cache the mapping of that item from responding to clients' requests). The authors also plot numbers for the temporal effects of the flooding attack. However, since the distributed service was largely unaffected, there is no great revelation there.

Next, the authors compare the replicated service to the distributed service. For the replicated service, the experiments try two versions: one where a random service replica is contacted and one where the best (closest to the client) service replica is contacted. The authors conclude that both replicated and distributed services are much more resistant to DoS attacks than the centralized service. The replicated service outperforms the distributed service when the optimal service replica is selected, because then some clients always avoid links congested by the attack, whereas in the distributed scenario, finding the appropriate service node may traverse congested links multiple times. The distributed service may become more appealing, however, due to its self-organizing nature.

For corruption, the authors found that routing attacks had minimal effect, and distributed service-specific attacks affected 24% of the network.

Based on these conclusions, the authors build a utility function to use in ranking the different service approaches, resulting in favoring the replicated directory service (with some anycast facility for locating the closest service replica) and then the distributed service.

39 Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance (David R. Cheriton and Timothy P. Mann)

[47]

What do you do when you want to actually use the name service much much much more frequently than [104] allows? Say, for global file name

service. Assume the existence and full deployment of multicast.

Three target classes of operations:

Binding assigning a name to an object, removing a name from an object

Query look if there are any objects assigned a name, if such a name exists

Mapping deliver a message to the object named by a name

General idea is to split the name hierarchy into multiple levels with different characteristics:

global Countries, types/groups of institutions. Members of this level are mutually distrustful. Enforcement of uniqueness is recommended (i.e., French sites can't be assigned names that appear among German sites). They change relatively slowly, so they can be heavily cached. They are extremely necessary for lower-level lookups, so high availability has to be ensured through replication. Performance is not that critical.

administrational Specific institutions, organizations. Members of this level might be mutually distrustful, but they respect the arbitration authority of the parent organization. They should be able to operate even when disconnected from the global context. More dynamic, localized information is represented, so a better, faster, more complete way of maintaining coherence is better advised here.

managerial Specific objects inside organizations, servers, handlers, managers. They only need to be available as long as the service they name is also available. They aren't as conducive to caching, because of rapid changes in their mappings.

At the managerial level, a name server is collocated with the object manager of the objects it handles. Once it receives a named request, it can immediately dispatch it to the right object. Similarly, when the manager is down, so is the name server. Getting to the right name server/manager from a client is the job of *prefix caches*, and realizing a name doesn't belong to any managers at all (i.e., it's a non existent name), is the job of higher-level name servers.

The prefix cache contains path prefixes (i.e., a number of path components) which map to specific managers. For example, the mount point of

an entire file server would be a path prefix and the server itself would be the specific manager. Even deeper prefixes can be accommodated, since for each entry, a manager identifier and a manager-specific directory identifier is returned. In the example above, for a subtree of the mount point of a file server, the file server's ID and the i-node of the subtree would be returned.

If the cache lookup is a *near miss*, i.e., it identifies a directory which is not a local managerial directory, then a probe operation on that directory is initiated. This just multicasts the request on all those managers covering (i.e., having mappings within) the local administrative directory. When one of them responds, the entry is put into the cache and operations proceed normally. If the near miss points to a name outside the local administration, then a *liaison server* is probed for the right managerial directory identifier. There's never a cache *miss*, since there's always a cache entry for the root global name servers. If that entry becomes stale, the client may resort to probing all nearby managers and liaison servers (through scoped multicast), trying to locate a new root entry.

When a cache entry becomes stale, then the server it was obtained from is asked to wake up and smell the coffee, which gives feedback to the server that it's caching stale entries. Stale entry detection is done either through timeouts (i.e., looking for a manager that's not there any more), or through appropriate error codes (for example, a manager replying that it's not managing the named object any more; for that to work, requests from clients have to contain both the [managerID, directoryID] tuple and the name they think maps to that).

The advantages here are:

- Even a near miss provides useful information
- Having short prefixes instead of entire names in the cache improves the hit rate
- On-use cache consistency checking might be slightly slower, but doesn't impose preventing cache consistency checks, which could be wasteful.

Administrational directory servers store, for each of their subtrees, sets of managers that cover the associated name space. Obviously, they are authoritative only for those prefixes that do not exist (i.e., those with a subtree name they don't have).

At the global level, designs presented for instance by [104] are used. Liaison servers become the front end to global level name servers, making it easier for clients to deal with fewer naming protocols.

To improve caching performance, *cache inheritance* is used. Parent processes bequeath their naming caches to their children.

For nearby objects, mapping only fails when the object manager itself fails or is inaccessible. Otherwise, (i.e., for objects outside the local administrative domain), mapping can also fail if all the administrative directory servers for the target organization are down and the client does not have their member managers cached.

Queries (i.e., binding checks and listings) are resilient against most managerial failures (since the administrational servers can answer them). Complete resilience can be guaranteed only if all clients are caching all global information.

Security covers:

1. Which clients are authorized to request information from a server
2. Which clients are authorized to update information on a server
3. Which servers are authorized to provide specific information

The reverse authentication problem in 3 above is the most difficult, since a client doesn't even know which manager it wishes to hear from, only that it should be "the right manager" for the particular query. This is customarily solved using a capability-like scheme of certificates. A certificate authority issues time-limited certificates for each server stating that "server X is authorized to answer query Y for time T". A server returns such certificates along with its signed answers. When a certificate expires, the server needs to get a new one from the certificate authority.

39.0.12 Issues

It's mentioned that object managers may choose to switch directory identifiers fairly often, and administrational managers may choose to switch the identifiers of managers fairly often (say for load balancing). This means that invalidation of cache entries can be a fairly frequent occurrence, unlike what's implied in the discussion in 3.2.

40 Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites (Ludmila Cherkasova and Peter Phaal)

[48]

This paper describes discusses then-current admission control strategies in overloaded web servers and identifies their preference for shorter sessions (i.e., sessions made up of fewer individual requests). This bias can be a problem in commercial web servers where there is a strong correlation between long sessions and sessions that result in a purchase.

The authors show through simulation that and overloaded server "prefers" shorter sessions. They go on to describe admission control techniques that, instead, minimize the number of aborted on-going sessions. The basic idea of the technique is to quantize time, and determine whether to admit new sessions during each time quantum based on how loaded the server was in previous quanta. A "responsive" strategy stops admitting new sessions if the utilization of the server during the previous quantum was above a threshold (e.g., 95% utilization). A "stable" strategy takes under consideration the utilization of several prior time quanta, only cutting down the admission of new sessions when several prior time quanta were above a threshold. Where the responsive strategy may overreact during short-lived request bursts, the stable one may underreact causing problems later on.

The authors proceed to tune these techniques into a hybrid approach, using as a "stimulus" the number of requests that are dropped from the end of the server's request queue within a time quantum. The server quickly switches to the responsive strategy when requests are dropped due to queue overflow and then slowly switches to more stable strategies as time passes without queue overflows. Finally, to combat the coarse granularity at which systems report utilization, the authors describe a predictive strategy that determines a "quota" of sessions that should be serviced during a time quantum. Once this many sessions are admitted, no more are for the duration of the time quantum. In combination with the admission control strategy, this predictive technique can help reduce the effects of highly bursty workloads.

An important part of this work is the design constraint of rejecting requests gracefully. Request rejection is not zero-cost, since it implies the

transmission of an HTTP response, which has a comparable cost to serving a small web page. As a result, it is important to take under consideration to cost of rejections for the analyses of all strategies described.

41 System Deadlocks (E.G. Coffman, Jr., M. J. Elphick and A. Shoshani)

[51]

The causes and repercussions of system deadlocks.

Resources subject to deadlocks can be tangible things (like disk channels) or abstractions (like a table or a process). There are resources that can easily be preempted (like the CPU) or not so easily preempted (like the disk). There are resources that imply the acquisition of other resources (for example files imply having buffers through which to access those files).

Conditions for a deadlock:

- Exclusive control of resources
- Resources can be claimed progressively; while new resources are vied for, others already claimed are held vehemently
- Resource acquisition cannot be preempted
- Circular chains of required resources exist (i.e., A wants something that B has, B wants something that C has, ... , Z wants something that A has)

Havender's approach to preventing all of the above from happening at the same time:

- Request all of the required resources at once. Don't proceed until all have been granted. (Break the second condition above)

  This can be costly, since some of the resources might only be needed temporarily.

- If, while holding some resources, a resource request is denied, give them all up and try again from scratch. (Break the third condition above)

  This can be really inconvenient when used on inherently non-preemptible tasks, like disk files.

- Impose an ordering in the types of resources, so that no circular request orders can occur. (Break the last condition above).

  This can be effective, although there can always be deadlocks caused among resources of the same type, especially when the instance of the type is not important.

The first condition above is considered unbreakable, since some things have to be granted in an exclusive manner (disk blocks etc.)

Detecting the occurrence of a deadlock can be as simplistic as forming a resource request graph occasionally and checking it for cycles, or as educated as monitoring everything every task ever holds and requests and deducing when the combination describes a deadlock.

Acting on a detected deadlock means forcibly reclaiming the resources held by one or more of the participants. The author presents a simple, cost-based scheme to pick which deadlocked tasks to abort, or even minimize destructive aborts (i.e., aborts that involve non-preemptible resources).

Another approach between making deadlocks impossible and just waiting until they occur to fight them is searching ahead into the state space of the system and identifying approaching deadlocks (for instance, two current tasks seem to be likely to form a circular resource request path). In those cases, through the scheduler for instance, the approaching deadlock can be avoided by modifying current system behavior (in the above example, making sure that the two dangerous tasks don't get scheduled in an interleaved manner until the danger has passed).

In summary, there are three ways to deal with deadlocks:

- *Deadlock Prevention* This means making it impossible for deadlocks to occur. Havender's approach above serves this purpose.
- *Deadlock Detection* This means identifying a currently occurring deadlock and breaking it by taking action against the impasse.
- *Deadlock Avoidance* This means detecting the *approach* of a deadlock and swaying the response of the system to avoid the danger.

41.0.13 The State Representation and the Detection Algorithm

r_1, r_2, ..., r_s represent resource types. w_1, w_2, ..., w_s represent the number of resources of each type. $p_{ij}(t)$ denotes the number of resources of type r_j

allocated to task T_j at time t. $q_{ij}(t)$ denotes the number of resources of type r_j requested by T_i at time t. Matrices $P = ((p_{ij}))$ and $Q = ((q_{ij}))$ define the allocation and request snapshots of the system, respectively, and row vectors P_i and Q_i are the allocation and request snapshots for task T_i respectively. $V = (v_1, v_2, \ldots, v_s)$ is the available resources vector, where $v_i \leq w_i$ indicates the number of resources of type r_i currently available.

At every level, the algorithm picks a task that requires fewer resources than those currently available. This means that, if all tasks take their chance to reserve the resources they require, this task will have its chance to advance and complete. After this determination, a task is marked "able to progress" and the resources it currently holds get added to the pool of available resources (since, either way, it will release them). When no more tasks can be marked "able to progress", and if there still exists unmarked tasks, then the algorithm concludes that a deadlock is in progress.

For an example, assume $W = (5, 8, 13)$, $V = (2, 1, 1)$, $Q = \begin{bmatrix} 2 & 2 & 6 \\ 2 & 6 & 0 \\ 1 & 1 & 1 \end{bmatrix}$, and $P = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 4 & 0 \\ 2 & 3 & 10 \end{bmatrix}$. In the beginning, the future available resources vector is equal to V, or $(2, 1, 1)$. The first pass of the algorithm locates the third row of the Q matrix, $(1, 1, 1)$, all of whose elements are less than or equal to the future available resources. Therefore, the third line of Q is marked, the future available resources vector becomes $(1 + 2, 3 + 1, 10 + 1) = (3, 4, 11)$ and the algorithm continues. Now the first row of Q fits the condition, so it is now marked and the future available resources vector becomes $(3 + 1, 4 + 0, 11 + 2) = (4, 4, 13)$. Finally, the middle row of Q can satisfy the marking condition and is also marked, leaving no unmarked rows and, consequently, signifying the absence of any deadlocks.

If Q had been $\begin{bmatrix} 2 & 2 & 6 \\ 2 & 6 & 0 \\ 3 & 1 & 1 \end{bmatrix}$ instead, then no line would be markable at the beginning of the algorithm, and we would therefore be in a deadlock.

42 Proactive Caching of DNS Records: Approaches and Algorithms (Cohen, Edith and Kaplan, Haim)

[52]

This paper explores techniques for dealing with the relatively high latency imposed on web-browsing by DNS queries. It models the cost of three proposed solutions, *proactive prefetching*, alternate cache replacement policies and *simultaneous validation*.

The authors argue that there is a lot to be gained from porting the results of content caching research onto DNS caching. On one hand, there are significant differences: DNS caching requires considerably less space per entry, less network bandwidth per request, and cache validation is as expensive as a fresh query. On the other hand, there are significant similarities as well: cache misses are very expensive, entries usually expire long before they have to due to change, request sequences exhibit reference locality and characteristic popularity levels.

In that vain, the authors propose a score of solutions "borrowed" from content-caching:

Proactive Caching This tries to strike a balance between the cost of cache misses and the overhead of issuing additional, potentially useless DNS queries.

Alternate cache renewal policies These explore alternate options in the policy used to maintain a DNS cache, based on reference locality (LRU), popularity (LFU), and adaptive, per-hostname policies. On a similar path, "prefetching" of sorts (called *preresolving*) is also examined, based on included hyperlinks on a page, for example.

Though the generic renweal policies mentioned apply at the DNS cache level, preresolving needs the cooperation of the web browser, and gives results specific to it. The two techniques differ in the effect on the shape of the traffic. Preresolving contributes more to a bursty behavior and is more likely to involve loaded root servers, whereas renewals access less loaded DNS servers lower in the hierarchy.

Simultaneous Validation This tries to exploit expired DNS cache entries. It uses them to begin fetching the intended content, while parallely

> requesting a fresh DNS entry. Once the DNS entry is available, and as long as it does not conflict with the expired entry, the already-present content is instantly displayed. Simultaneous validation does not impose overhead of additional DNS queries, although it imposes a closer cooperation of a web-browser with the cache of the DNS resolver, and requires the resolver to maintain expired cache entries.

To further motivate their research, the authors have looked at a limited sample of DNS queries generated by web-browsers. Based on a preliminary study of web-cache patterns, they have found that most host names used in browsing require at most two distinct name server requests, starting from the root name server. Furthermore, they've found that TTL values are in general quite conservative: about 90% of the hosts in their sample have TTL values no greater than a day.

One data point that can motivate modifications at the cache level is that quite a few name servers serve many distinct zones, which means that the usefulness of their caches can be quite high.

43 An Analysis of Wide-Area Name Server Traffic: A Study of the Internet Domain Name System (Peter B. Danzig, Katia Obraczka, Anant Kumar)

[55]

A major difference to current trends in this paper is its pre-web approach to Internet wide traffic. They say that the predominant source of request/response traffic on the Internet comes from name server traffic. This is not true any more in post-web times, in all probability.

The basic issue dealt with here is how DNS was flawed, how that led to performance problems, and how the problems could be solved.

One of the implementation and deployment issues observed is the predominance of collocated caching-only servers with client resolvers. The absence of centralized caches within a site, that could share name caches among multiple local resolvers, leads to multiple queries issued for the same result from the same site.

The paper includes a technique for calculating the packet loss rate on a single ethernet, given the numbers of unmatched requests and responses for local queries.

It seems that the paper is less applicable to directory services other than the DNS, especially since it tends to focus on DNS bugs and how they have been manifest (for example, some resolvers' mistake of adding the local domain name to all queries, assuming client programs always gave them unqualified host names).

An interesting contribution of the paper is a classification scheme for server errors (bugs, incorrect assumptions, etc.). One of the conclusions drawn is that negative caching is in general overstated, mainly because it is intended as a safety valve for faultyimplementations of servers or resolvers.

An interesting data point is the amount of replication per directory unit in a large set of servers. The authors use these data to support "unfair" leak of performance degradation between multiple domains serviced by the same server.

The authors claim that active error detection is more important than widespread deployment of bug free implementations. The reason is that bugs may resurface, despite the efforts of programmers. Active error detection can limit the impact of current or future bugs. One proposed technique is "playing dead". According to it, a name server can occasionally play dead for a particular queried name and record the requester's reaction. This can show problems like untuned retransmission intervals, for example. Another approach would be to keep track of how many times the same query has been submitted by the same client host. Erratic hosts can be identified in this way and notified of their errant ways.

44 Compliance Defects in Public-Key Cryptography (Don Davis)

[56]

This paper deals with the hidden, painful assumptions lying behind the practical use of public-key cryptography. The two important requirements that go hand-in-hand with public-key cryptography are, first, that public-key certificates must be validated against tampering and revocation rigorously and, second, that private keys must be kept secret and secure for at least

the duration of their validity, which frequently extends to a year or longer. The difficulty of satisfying both requirements supports the author's thesis that public-key cryptography is better suited for server-to-server interactions, where trained, professional administrators can be trusted to follow rules, whereas asymmetric cryptography is better for layperson end-users.

The author defines the term of *compliance defect* to cover those rules of operation that are difficult to follow and that cannot be enforced. He addresses five such defects in the paper:

- Issuance: How does a CA authenticate a distant user when issuing an initial certificate? This defect seems to apply to both public-key and symmetric-key cryptography systems, though, as the author admits in section 5.

- Validation: It is difficult securely to distribute and revalidate with time the root public key. A way to deal with viruses of the trojan variety is to keep the root public key signed by the local user; a virus cannot modify the root public key without first knowing the local user's secret signing key.

- Revocation: Since no central authority takes part in uses of public key certificates, there is no choke point where revocation checks can be enforced. Instead, users themselves must check applicable certificate revocation lists before every use of a correspondent's or of CA's public-key certificates.

- Single sign-on: The user's private key must remain in main memory while the user is logged into a particular computer, for convenience. This makes that computer extremely vulnerable when the user is not taking care of it physically.

- Password change: Combined with the physical security of an encrypted private key, the quality of the pass phrase used to encrypt that private key is of paramount importance. This is especially critical when the user maintains his private files, including the encrypted private key, on remote storage.

The author makes the interesting suggestion that public-key cryptosystems are, nominally, so much better than symmetric ones because they transfer the burden of diligence from the server components to the users. Specifically, whereas a Kerberos [160] KDC (Key Distribution Center) needs to be

highly available to issue tickets and change passwords, a CA need only be there long enough to issue certificates; all further certificate distribution can be done by users themselves or via an untrusted certificate directory. With recent work on certificate revocation trees [99], the situation is similar for revocation. However, whereas a KDC can check the quality of passwords set by a user, no such tool exists to force vigilance for users' pass phrases. Similarly, a user can ignore revocation checks if he wishes, whereas a KDC is less accommodating.

The author hints at hybrid symmetric/asymmetric systems as a means of dealing with the problem, although the specifics for such systems are described elsewhere.

45 Using Client Puzzles to Protect TLS (Drew Dean and Adam Stubblefield)

[57]

The goal of this paper is to explore client puzzles as a technique to prevent denial-of-service attacks against secure transport protocols, specifically TLS.

The problem stems from the fact that the TLS protocol requires an RSA decryption from the server with every incoming connection request. This allows an attacker to swamp a TLS server without expending significant amounts of work or bandwidth, since currently a TLS server can handle very low RSA decryption rates.

Instead, the authors suggest the use of a client puzzle and the "price of admission" into the TLS protocol. The type of puzzles they use is hash-function inversion. Using a strong pre-image resistant hash function, such as MD5 and SHA1, a server poses a puzzle that a client has to solve before being able to participate in the main portion of the TLS protocol. The puzzle looks like $(n, x', h(x))$, where $h(.)$ is the strong hash function, and x' is the same as x, except all n of its low-order bits are zeroed out. The puzzle essentially invites the client to brute-force the hash function so as to discover those n low-order bits of x. This should take 2^{n-1} hash operations at the client. At the server, all that is needed is the generation of a random number (x, a random 512-bit data block) and the evaluation of the hash function on x, which are both reasonably cheap operations, certainly cheaper than an RSA decryption.

Client puzzles are only used when a TLS server is overloaded, to throttle new client connections. When the server is not overloaded, the puzzles are unnecessary, since they increase latency and limit the clientelle only to those clients who support client puzzles. Server load is determined by counting pending RSA decryptions. When the backlog of RSA decryptions exceeds a certain threshold, the server starts requesting client puzzle solutions before accepting another RSA decryption into its backlog. There is a different, lower threshold under which the server stops requesting client puzzles. The purpose of the disparity between the two thresholds is to reduce oscillation between client puzzle mode and normal mode.

Experiments run by the authors show that an unmodified TLS server attacked can cause legitimate clients' latency to jump up by 2 or three times. With a client puzzle-enabled TLS server, client latencies remain low and unchanged during an attack.

46 The Bayou Architecture: Support for Data Sharing among Mobile Users (Alan J. Demers and Karin Petersen and Mike J. Spreitzer and Douglas B. Terry and Marvin M. Theimer and Brent B. Welch)

[58]

This position paper takes an overall look at the goals of and resulting requirements from a data sharing environment for mobile users. Disconnection is a fact of life, so an effort is made to make the design highly usable during disconnection.

- **The system must support mobile devices have limited resources** This leads to a design where the line between clients and servers is practically invisible. A mobile device might act as a client to retrieve information from a remote device, or as a server, to propagate updates to some other device. This setting has been commonly referred to as a *peer-to-peer* in the early 2000's.
- **The system must offer high availability for reads and writes** This leads to a weakly consistent system. Strong consistency does not

allow high availability when disconnection is frequent and the size of the disconnected partitions is invariably smaller than a reasonable quorum.

- **The system must be able to reach eventual consistency in reasonable communication settings** This leads to the good old anti-entropy scheme for update propagation (a.k.a., gossip, rumor mongering, etc.) Anti-entropy allows updates to permeate the system even if some replicas never communicate directly with each other.

- **The system must detect update conflicts** Although write/write conflicts (when two clients update the same item in different partitions), and read/write conflicts (where a client updates an item in a partition based on a read value that a client in another partition is currently updating) can be reasonably dealt with using version vectors (e.g., as in Locus [176]), they ignore application semantics. This requirement leads to a design where every write is accompanied with a dependency check that verifes whether the write causes a conflict or not.

- **The system must support application-specific resolution of update conflicts** This leads to piggybacking data updates not only with a conflict detector but also with a conflict resolver.

- **The system should commit tentative data values as soon as possible** This leads to the election of a primary server per data item. Once the primary for a data item has accepted an update, that update is no longer tentative, and can propagate back into the system.

- **The system should allow disconnected clients to see their own updates** Although disconnected updates are tentative if the primary for the affected items is not in the partition, they can become committed if no other entity outside the partition conflicts with them.

- **The system should allow clients to have a view of the data that is consistent at least with their own actions** This naturally leads to the session guarantees described by the authors previously[171].

- **The system should allow applications to specify how consistent they need their view of the system to be** Other than allowing each application to specify its own session guarantees, this also means

allowing applications to choose whether to see tentative or committed data, how old the data they see are, etc.

- **The system should allow a flexible, controllable placement of its replicas** This leads to a dynamic scheme for replica creation, use and movement, or *fluid* replication.

All of these design decisions necessitated by the goals posed by the authors have been pursued within the Bayou project.

47 Data Security (Dorothy E. Denning, Peter J. Denning)

[59]

Internal security (as opposed to external: armed guards, fire protection and such) is mainly focused on the following:

- access controls
  - It relies on being able to identify principals securely (authentication).
  - It makes temporary observation awkward.
  - The privilege database is extremely sensitive and should be kept out of reach.

  There are two major design patterns:
  - Access control lists
  - Capabilities
- flow controls

  They limit the information flow to prevent leaks from secret to less secure classification. Once you're at class A, you can't talk to a lesser class. This scheme incurs severe overclassification, since all principals end up going up in classification. It doesn't model the world (for instance, NDAs temporarily increase a person's classification for a certain domain, with specific penalties in case of privilege incursion)

To fix, info flow must be deduced (which input affects which output). Detect covert channels. This is very very hard.

- inference controls

  Restricted private info can be inferred from unrestricted aggregate info: ask controlled question when part of the answer is known (trackers).

  Solution: reduce accuracy of answers, use random samples.

- encryption

  It helps with authentication, guards against eavesdropping.

48 Working Sets, Past and Present (Peter J. Denning)

[60]

A painful paper, with no less than 126 references...

A program's working set is the collection of pages it has recently referenced.

A bit unclear what exactly Saltzer's Problem is. A box with a knob we turn to tune performance? Not very clear. The knob here is the control parameter θ, which governs the tradeoff between resident set size and longer times between faults.

The swapping curve plots $f(x)$, where x is the mean resident set size and f is the rate of page faults. The lifetime curve plots $g(x) = 1/f(x)$, i.e., the mean number of references between page faults. The global max of $g(x)/x$ is the best a policy can do. Simple models for g fail to capture the behavior of real programs.

A restatement of the policy: *The working set policy is that under which a program's resident set is the collection of pages that have been referenced in the last θ time ticks.*

A generalization deals with the "retention cost" of a segment, accumulated since its prior reference. The generalized working set policy retains pages whose retention cost doesn't exceed a certain limit.

Simple program behavior models fail to capture phase transitions (they assume locality characteristics remain constant). "Slow-drift" models are

a compromise: they assumed a continuous degradation of locality. Real measurements proved these inadequate as well. In practice, slow-drift can only account for program behavior within short phases of its execution.

The phase/transition model specifies that long high-locality periods are separated from each other by short, erratic transition periods. Phases cover about 98% of the time. Transitions cause about 50% of the page faults.

Network queueing models have been used to determine which is the optimal "multiprogramming level" (load) of a paging system. Studies have shown that the "$L = S$ criterion" (when the time between faults L is equal to the time it takes to service a fault), or the "50% criterion" (keeping disk utilization to 50%) yield close approximations to the optimal load.

The problem with the above criteria is that they attempt to apply to all programs, even when for example the max lifetime doesn't exceed the page-in time. Space-time minimization per job should be the optimality criterion. Load controls (limits on the load of the system) have been used to limit the drop-off in CPU utilization when thrashing ensues.

The basic components of a dispatcher are:

- the scheduler, which activates or deactivates jobs and allots them a time to use the CPU.
- the memory policy manages which pages will remain in memory, either using local criteria (like WS) or global (like LRU).
- the load controller adjusts the limit on the load of the system based on current conditions, globally or locally.

For local policies like WS, the selection of θ for each job is a daunting task. However, it was found that for (allegedly) typical program mixes, a global WS θ yields performance within 10% of the optimum. Furthermore, WS performs the exact same sequence of page faults as the optimal lookahead method, displaced only by the lookahead time. It is very close to optimal, when fully tuned.

49 The Sybil Attack (John R. Douceur)

[61]

This paper take a formal approach to the problem of identity establishment in a large-scale peer-to-peer system. The thesis of the work is that,

without a central authority issuing identities, it is practically impossible to establish the one-to-one correspondence between an identity (a logical identifier) and an entity (a real-world agent). Being able to determine whether or not a number of identities belong to disparate physical entities is essential to any Byzantine fault-tolerance scheme based on redundancy.

The author uses a simple formal model of the infrastructure on which most practical peer-to-peer systems rely to prove his claims. The model allows entities to communicate reliably (though not necessarily in order) with all other entities via broadcast. *Correct* entities play by the rules of the established protocol, whereas *Faulty* entities may act arbitrarily according to a plan or randomly. This is a more permissive model than what actually is possible on the Internet, since the Internet operates on a best-effort basis, which means some messages never get delivered at all.

The minimal restriction placed upon entities involves the resources available to them. Specifically, entities have computational resources that are polynomially bounded.

The objective of the system is to prevent a misbehaving, faulty entity from successfully authenticating itself to another that via two or more apparently distinct identities.

The results of the paper start with a lemma showing that an entity can successfully authenticate itself via a number of identities that is proportional to the resources available to it. Specifically, if the faulty entity holds resources that are ρ times those of the minimal necessary resources for identity establishment, then the faulty identity can authenticate itself as at least ρ identities.

One way that the authors proposes to combat the problem is by making this proportionality an upper bound. This can be accomplished if all authentication is resource-bound: I can barely authenticate my identity if I use up all of my (reasonably high) bandwidth, I can barely authenticate my identity if I use up all of my (reasonably high) storage, and I can barely authenticate my identity if I use up all of my (reasonably high, polynomially bounded) computation.

The second lemma in the paper goes further to prove that if identity authentication can be performed at arbitrary points in time, then a faulty entity can still authenticate itself as an arbitrary number of apparently distinct identities. The idea behind this is that if my authentication is resource based, then I can use all my resources to establish my first identity, then use them all again to establish my second identity, and so on. Storage seems to

be the only kind of resource that seems harder to adapt to such an approach. As a result, the author claims that all authentication in the system must happen concurrently.

Similar results are presented when identity authentication can also happen indirectly, i.e., thorough introduction by a large number q of established identities. Here the author shows that if the set of colluding faulty nodes has more resources than those available to q legitimate entities, then the faulty set can establish arbitrarily large identities. Even worse, unless indirect identity authentication is also done concurrently, any single faulty entity can also establish an arbitrarily large number of identities, by establishing distinct identities with different groups of honest entities one after the other.

The author concludes that the assumptions for making server-less identity establishment possible (identical resource constraints, simultaneous identity authentication, relatively small percentage of possible faulty nodes), a centralized identity service is necessary.

50 Introduction to Personal Communication on the Internet (Christian Dreke)

[62]

This is an overview of the current available modes of personal communication (on the Internet).

It goes over communication media both in the real-time category (instant messaging, text chat, internet phone, and collaborative/whiteboard applications) and in the asynchronous category (email, newgroups, and shared spaces).

Then the author proceeds to present the application contexts within which these communications media are combined into personal communications applications. Buddy lists are systems where a person subscribes to his circle's members; when a member "comes on-line", the subscriber is notified. This is the general framework within which On-line presence work is being conducted. Other such contexts include internet telephony, virtual communities (e.g., multi-player gaming, real-time newsgroups/topicgroups), and virtual worlds.

The main barrier preventing these applications from becoming predominant is standardization, especially in the Instant Messaging/Presence arena:

there are too many proprietary protocols striving to secure a market share, completely precluding interoperability among all users' software.

51 Establishing Identity Without Certification Authorities (Carl M. Ellison)

[64]

This is one of the first papers that attacked the notion that public key cryptography absolutely requires a PKI, a public key infrastructure. Ellison looks at the main building block of PKIs, the *identity certificate*, which is a statement by a trusted certifier that binds a name to a public key. He argues that identity certification is almost never necessary in common transactions we expect to perform online, compared to *attribute certification*, that is, the certification that the owner of a public key has a particular attribute, such as a particular face, has access to a particular bank account, etc.

Identity throughout the paper is regarded as the "distinguishing character or personality of an individual" (from Webster's), that is, what makes a particular person an individual, as opposed to someone with a particular name. When, as a verifier, I wish to establish I'm talking to the person I was talking to before, I care less about that person's name and more about his sameness to the person I was talking to before.

Ellison goes over different kinds of identity certificates. X.509 certificates bind keys to X.500 names [91]. However, X.500 names are used nowhere other than in X.509 certificates, so they are not very useful. Furthermore, X.500 names consist of disambiguating information, such as the organizational unit in which the named individual belongs, that a correspondent might not necessarily know beforehand[2]. PGP certificates use email addresses, which are more useful that X.500 names, but are issued not by companies or governments but by individuals, who might not be widely trusted. Driver's licenses are used as identity certificates some times, although their function does not require name binding; instead, it requires either that a particular person can be identified uniquely (a serial number would do), or that certain attributes (height, image) are associated with a signature, in which case the name on

[2]To this I add the privacy problem inherent in X.500, since X.500 names expose information that shouldn't necessarily be public, such as the org-chart of a company.

the card is irrelevant [3]. The gist of this section seems to be that identity certificates are used in many online and offline tasks, even though a name-to-key binding is quite often unnecessary in those tasks.

Then Ellison describes ways to use certificates that are not issued by a central authority in different relationship contexts: personal contact, major branded corporation, network contact, complete strangers. In all examples, a central certifier is, if not useless, at least surmountable.

The last portion of the paper tries to address the problem of exchanging public keys over a public, unsecured communication medium when the two participating parties only know a limited number of real-life, low-entropy shared secrets. The goal of Ellison's solution is to thwart man-in-the-middle attacks, where a malicious adversary tries to insert himself between the two parties and intercept (perhaps even modify) exchanged communication.

The two parties create fresh key pairs and send the public portions to each other (those public keys may be intercepted and replaced by the adversary). Then they proceed to exchange challenges (e.g., what was your favorite food in 1979) and low-entropy, presumed secret answers to those challenges (e.g., "french fries" or "fries" or "chips"). After a while, the participants either determine that their conversation is being manipulated, in which case they abort and black list the secrets they exchanged as compromised, or are convinced that a man-in-the-middle attack is highly unlikely.

At every round, participants exchange questions and then hashed (cryptographically) answers, keeping track of the entropy of valid answers to their own questions. If after a few rounds of successful responses to the challenges from either side, the sum of answer-pool entropies has become high enough, then the participants can decide that an adversary is unlikely to have been able to guess correctly every single time. If a challenge is not responded to, the protocol stops and communication is aborted.

To protect the low-entropy secrets from being brute-forced through dictionary-like attacks, Ellison suggests a blinded interlock protocol that releases answers to challenges one bit at a time, giving a potential adversary only little fodder for dictionary attacks.

[3] Here it seems obvious to counter that serial numbers are identifiers, making the license an identity certificate still, only from a different name space.

52 The MULTICS Input/Output System (R. J. Feiertag, E. I. Organick)

[66]

One of the earliest suggestions of device drivers, device-independent I/O interfaces, standard I/O, redirection.

Device-independent I/O was made possible by the memory mapped nature of Multics. Once a segment was made known (no matter what device it resided on), it could be accessed independently of its implementation or capabilities.

Device-specific data manipulations are done inside the "Device Interface Module", which is pretty much an object-oriented device driver in current terms. Its interface even provided for device-specific commands, similar to ioctls.

A very exciting paper to read, given the context in which it was written.

53 Impossibility of Distributed Consensus with One Faulty Process (Fischer, Michael J. and Lynch, Nancy A. and Paterson, Michael S.)

[67]

The point of this sort of elusive paper is to prove that under a specific set of asynchrony assumptions, a distributed consensus protocol is impossible to define, even if only one of the participants is allowed to be faulty.

Asynchrony is defined as the ability of eventual delivery of all sent messages to their intended destination. In other words, a message always takes a finite number of time units to be delivered to its destination, but that number if time units is not known to be bound in any way.

Actually, the results in the paper are proving the impossibility of an even weaker result: that such a distributed protocol can always lead to a decision made by at least one of its participants. The proof first assumes that a partially correct protocol exists that can always yield a decision, and then moves to reach a contradiction. *Partially correct* is a protocol in which

- when a decision is made by a process, that is the only decision made

- any possible decision can be reached from at least one valid initial state

The definition of a *faulty* participant only includes fail-stop situations, i.e., doesn't even begin to encompass byzantine failures. A *non-faulty* participant is one that is never unable to process inputs.

The proof begins by showing that in any ideal consensus protocol (i.e., that's partially correct and is resilient in the face of a single faulty member), there is at least one initial state from which any possible decision can be reached (that's not the same as the second clause in the definition of partially correct above; this step shows that any possible decisions is reachable from the *same* initial state; the clause in the definition above only states that there are initial states which can lead to any possible outcome).

The second stage of the proof shows that, starting from a state where any decision is still reachable, any of the events that can be applied to that state (an event is a message received by a participant), can hold off long enough, so that when it is finally applied, the state of the protocol can still lead to any possible decision. In other words, if at state C event e is applicable, then there is a sequence of steps the protocol can take, after which, applying e, the resulting state can still lead to any possible decision. This second stage takes under consideration the requirement that a single faulty participant must be tolerated at any step of an "ideal" protocol's run.

Finally, these two stages are combined to show that it is possible to run the "ideal" protocol, starting from an initial state from which any decision is possible, and keep arriving at intermediate states from which any decision is also possible, without ever running out of things to do for any of the participants. This flies in the face of the original assumption that such an "ideal" protocol can always reach a decisive state.

The authors conclude by describing an alternate set of assumptions that can lead to a decisive protocol. Namely, they show a protocol that always leads to a decision, as long as all faulty participants are known right from the start, and no more than half of the participants are faulty.

This is a very dense paper, where many of the intermediate steps in the proofs are skipped. It is however very illumiating to see how theoretical impossibility results such as this can be derived.

54 Adapting to Network and Client Variability via On-Demand Dynamic Distillation (Armando Fox,and Steve Gribble and Eric A. Brewer and Elan Amir)

[70]

Basic motivation: networked devices are very diverse in capabilities. How can the same content be delivered and used at all these different devices, without being unduly duplicated or delayed?

Axes of variability:

- Screen size
- Bandwidth
- color depth
- processing power
- available content encoding capability
- (not mentioned) power consumption

General idea: perform lossy on-demand content type conversion, to minimize network latency and bandwidth requirements. This goal can be achieved through the following three techniques:

1. Type- and semantics-specific lossy compression (*distillation* and *refinement*). The specialization of the compression to fit the semantic content under consideration can produce better results than generic compressors can.

   The semantic type (text, image, video) and the encoding (postscript, mpeg, ...) are logically independent. It is the content that's distilled, to reduce detail or quality (e.g., image resolution or color depth), but some operations are easier in some encodings than others. Refinement, on the other hand, allows the user to decide whether the full-blown, "expensive" version of some content should be downloaded, by looking at the "cheap", distilled version. Refinement amounts to an informed zoom-in operation, in terms of quality.

2. On-the-fly compression, though it can add some latency due to the computation involved, can eventually reduce latency, since compression results in less data to transfer.

   Besides, on-the-fly compression can also be customized to the particular needs of the client that initiated it (as opposed to using pre-distilled stored versions of the original content, and pigeonholing clients to available versions).

3. Removal of compression/adaptation mechanisms from the clients and servers results in happier servers (faster for what they perceive as the common case, since they don't have to waste cycles on adaptation, or space for multiple versions of content), happier clients (less memory/storage/power wastage), and richer proxy business owners. Besides, legacy servers can still serve clients with unique limitations.

Current techniques for dealing with client variability include providing multiple versions of the content (which means a bigger administrative mess and still leaves some requirements out), or just supporting one side of the fight (i.e., either the low end clients or the extra-heavy-duty high-end clients). Even when coarse-grained refinement is allowed (as is the case with web browsers supporting progressive image encoding techniques), it's done in a way that ignores semantics, or value of different parts of the conent, thereby treating bundled content indistinguishably.

The basic components of the proposed architecture are as follows:

Proxy It lives on the boundary between the slow and the fast network (or the two networks displaying heterogeneous characteristics). It figures out what needs to be done to turn the content into what the client can experience, and calls the appropriate distillers.

Distillers These provide the actual smarts of converting contents of different semantic type to fit the needs of clients.

Network Connection Monitor This keeps track of changes in the condition of the connections between a proxy and its clients. When the link status (effective quality) changes, the proxy adapts its conversions to fit the situation optimally. It can also be told what the conditions are by the user.

Client-side support Proxy-aware applications are linked with a special library, communicating with the proxy through a standard API. Proxy-unaware applications are using a "client-side agent", which fakes the API of a server to them, and then talks to the proxy.

55 Harvest, Yield, and Scalable Tolerant Systems (Fox, Armando and Brewer, Eric)

[69]

This paper introduces the thesis that in a real system you cannot have consistency, availability, and partition resilience, all three at the same time. The authors call this the *Weak CAP principal*.

One interesting position is that even systems which profess to be highly available, for example in the face of at most a single failure, are in fact probabilisticly available, since there is a non-zero probability that more than a single failure will occur. Modeling systems using probabilistic techniques is very important in understanding how real, practical systems work.

The authors propose the concept of *harvest* and *yield* to model the requirements of distributed applications. Yield is the probability of completing a request. Harvest is the fraction of the data reflected in their response, in other words the completeness of the answer to the query. Wide area systems can become more scalable when harvest and yield can be tolerated when they gracefully degrade. Degradation of yield means probabilistic availability. Degradation of harvest means lower fidelity of the results to the results that would be available in the best case, in the ideal operational conditions of the system.

Another interesting point brought up is that of orthogonal decomposition. By reducing unnecessary interactions between components making up the larger system, different selections for harvest and yield can be assigned to different sub components thereby offering a more flexible set of operational options for the overall system. Besides, it has been argued that orthogonality of the sub components of a system can lead to lower complexity and fewer bugs.

Overall this paper doesn't claim to fully understand the transition from the weak CAP principal to the concepts of harvest and yield but suggests that this would be a useful and fruitful research direction.

56 Automatic Inference of Models for Statistical Code Compression (Christopher W. Fraser)

[71]

The idea behind the whole thing is being able to provide compression algorithms with better statistical models of code, so that they can improve compression efficiency.

This is important, since code compression is becoming very attractive, both to reduce transmission speed of code over a network, and also to reduce access times from secondary storage and footprint.

The statistical model of a chunk of code is useful, because it determines how output symbols are assigned to input symbols: frequently occurring input symbols are assigned shorter output symbols, since the overall goal is to minimize the length of the output symbol stream.

The objective of a modeller is to output a bunch of *contexts*, which are identifiable states in the linear scanning of a code stream where symbol probabilities are relatively stable, and their associated *probability distributions*. To determine such contexts and probability distributions, a modeller checks a specific set of *predictors*, such as stack lengh, instruction under the cursor, etc., to output something like the following decision tree, which for each decision through the predictor set, yields a probability distribution:

```
if instruction=comparison
then use distribution_that_favors_branches
else if stackHeight=1
then use distribution_that_precludes_binary_operators
else use other_distribution
```

These decision trees are created by a simple algorithm that attempts to minimize the total entropy of the code stream. Implementations of the algorithm are still slow for on-the-fly model inference, but they are very useful for long-term compression (such as that used before code is stored on stable storage).

Commonly used predictors are:

- The last few (10-20) tokens seen. This catches cases like comparisons followed by branches, or additions of 1.

- Computed predictors, like the stack height or the data types of the last few operations.
- Projected predictors, which extract equivalence classes from specific instances (for example, projecting all different comparison operators like `EQ` or `GT` onto a single, `REL` operator).

57 Dos and Don'ts of Client Authentication on the Web (Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster)

[72]

This paper accumulates previously-unpublished experience with building client authentication schemes on the Web, either through the use of SSL-based transports, or without. The authors have surveyed a large number of "secure" web servers offering services to clients that are either membership-based (once a client is authenticated as worthy of receiving service, it doesn't matter who that client is), or personalized (the identity of the client is necessary for all interactions, since the server's responses are customized). They have found quite a few "home-brewed" authentication systems that failed to enforce policy, by either allowing adversaries to claim identities they shouldn't be allowed to claim, or by completely exposing the security machinery at the server. In response, the authors propose a list of principles on which a web-authentication scheme should be used, and explain one design that follows those principles.

Here is a summary of those principles (or "hints"):

- Use cryptography appropriately
  - Over-design (and therefore, overprovisioning in terms of how much security is required and used) can lead to complex systems, which in turn are hard to keep correct
  - Home-brewed cryptography is almost never solidly built. It is better to rely on well-studied, published cryptographic techniques, than to spin new ones.

  - Obscurity is almost never equivalent to security. A secret protocol is not secure just because it is secret.
  - Cryptographic tools have an intended use and a particular set of security guarantees. Just because a tool has been proven secure in its intended application does not mean that it is secure in another application for which no analysis has been done.
  - Composing security schemes is not always as secure as even the least secure of the schemes. Strange interactions between the composed schemes can lead to overall insecurity. Composition should only be used where it has been analyzed and deemed applicable.

- Use passwords appropriately
  - The most secure scheme in the world is wide open if the secret used in it, the password, is exposed. Passwords should remain secret for the duration of their validity.
  - Passwords should be resistant to dictionary attacks. Even a well-protected password is useless if it is a popular, easy-to-guess word such as "sex".
  - Password (and account information) changes should always be reauthenticated at the time of submission. This prevents a prowler from finding an unattended web browser that is already logged in and changing the password on that account without knowing the current one.

- Use authenticators carefully
  - Authenticators, such as capability-like cookies in the web setting, should be unforgeable (i.e., unguessable). If a secure hash function is applied on a set of secret information, it should be made certain that the different pieces of secret information are unambiguously separated from each other.
  - Authenticators that must remain secret should never be exposed to eavesdroppers. For example, if a secret authenticator is put in a web URL as a CGI argument, it should never be allowed to appear in referrer logs of subsequently accessed, unrelated web servers.

- Authenticators should not be put in persistent cookies. Persistent cookies live after the end of a browser process, and can therefore be accessed by different users of the same machine.
- Authenticators should not rely for their lifetime on the compliance of a client browser to cookie lifetime maintenance. The server should maintain the lifetime of the authenticator, preferably as part of the information encoded unforgeably within the authenticator.
- Authenticators should be difficult to reuse by eavesdroppers. One way to make such reuse harder is to bind authenticators to specific browser addresses. Of course, address spoofing can eliminate such safeguards.

The authors then proceed to describe a straightforward design that follows all the guidelines given above.

58 Public Key Distribution with Secure DNS (James M. Galvin)

[73]

This paper addresses the problem of key distribution for users, as opposed to hosts, on the Internet. It spends a great deal of time introducing the then fledgling secure domain name service (DNSSEC) [63].

The new information included in this paper, besides the explanation of how DNSSEC works, is a preliminary proposal for mapping user names to domain names. Specifically, email addresses are picked as the canonical human user name. Their mapping to a domain name is straightforward by replacing the character with a dot. For email addresses that contain characters other than the valid alphanumeric DNS characters, other options are explored, including replacement by dots or translation into an alphanumeric code.

The author includes in the paper some insights on how turning DNS into a key distribution architecture may involve shifts in the way we think about DNS. In the original DNS [118] parent zones need have no administrative association to their child zones; a parent zone merely points to the servers responsible for all of its children zones. In contrast, secure DNS places parent zones on the certification path from the globally trusted root to their

descendant zones, implying that parent zones are more trusted than their children zones.

Furthermore, to circumvent the pains of incremental deployment, secure DNS prescribes that zones carry a signed pointer to the public key of their parent zone. Unfortunately, no check is standardized that establishes that the pointers from parents to children and vice versa are consistent. Therefore, clients may be diverted to impostor parent zones of their local zone's choosing, which can be dangerous.

Finally, the secure DNS does not specify under what policies the use of a particular key is governed. This is an important issue in a secure architecture.

59 Weighted Voting for Replicated Data (David K. Gifford)

[74]

Replication of files in a distributed, transaction-enabled environment. The main points are:

- Every replica i is assigned a number n_i of votes.
- Every transaction collects r votes to be able to read a file, and w votes to be able to write the file.
- A read quorum and a write quorum must always overlap, i.e., $r + w > \sum_i n_i$
- Read/write quorum overlap guarantees that every read quorum contains a current replica of the file (i.e., one with the latest modifications in it).
- Current replicas of a file are determined using version numbers, which are increase every time a transaction starts writing into a file.

Votes are assigned to a replica depending on the reliability of the host on which a replica resides. A very very reliable machine causes replicas living in it to have many votes. When r for instance is lower than the number of votes of a very reliable replica, then reading that replica successfully is good enough. Obviously, reliability must also cover consistency in this regard.

Caches of a file can be maintained close to a client, by creating a copy with 0 votes. They don't participate in the consistency considerations, except as far as their version number is concerned.

Updates to the vote configuration are supported exactly like a normal file modification (every replica contains a directory of all other replicas, which is 1-1 associated with the version number of the replica).

60 Global Internet Trust Register

[5]

This book contains fingerprints for some of the most important public keys used on the Internet at the time of its publication (early 1999).

The reasons for putting the functionality of a global CA into paper form have been to

- offer a reasonably neutral top-level CA; commercial ones tend to avoid talking about their competitors
- capitalize on our instinctive trust of print over electronic publishing
- use the political privileges of print publishing, over electronic publishing, in the face of proposed government licensing or control of CAs.

X509 and PGP key fingerprints are included. The keys have been given a level trust, ranging from D (minimal trust) to A (full personal trust).

In the process of accumulating the keys printed within this book, the editors have identified some ominous problems in the way that certificates are handled in web browsers. They have unearthed implementation flaws (giving incorrect results), design flaws (ignoring important protocol-mandated fields from certificates), distribution flaws (hard-coding root CA keys into the application, whose lifetime might surpass the lifetime of the hard-coded CA key).

61 Implementation of the ficus replicated file system (Richard G. Guy and John S. Heidemann and Wai Mak and Thomas W. Page Jr. and Gerald J. Popek and Dieter Rothmeier)

[76]

This was an ambitious project attempting to create a massively large scale distributed file system. The claims to fame of the ficus replicated file system where stackable layers, similar to the whole V node approach, to promote flexibility, locality based caching, replication to increase availability, non-serializable consistency to increase availability in the face of partitions, and volume grafting to promote file mobility.

61.0.14 The ficus layers

The V node paradigm is used here because of its widespread availability and compatibility with current systems at the time of this work. However, the authors found the V node interface rigid and hard to extend. NFS was used as the transport layer and UFS as the physical disk storage layer.

The ficus functionality itself is split into two distinct layers, the logical layer and the physical layer. The ficus logical layer presents a single file copy abstruction to higher layers in the stack. It manages replicas internally, deals with update propagation, reconciles conflicting directories, etc. The logical layer assigns a file handle to the ficus file which is unique across an entire ficus system.

The ficus physical layer deals with the actual physical storage of a ficus file replica. Both files and directories in ficus are stored as UFS files. Additional metadata are stored as separate UFS files. Mapping from a ficus file handle into a UFS path name is done by turning the file handle into a hexadecimal string.

61.0.15 Replication

Communication interruptions are problems against which ficus replication fights. File handles are composed of a file identifier which globally identifies

uniquely a logical file, and a replica identifier which uniquely identifies the particular replica. Apparently, clients determine which and how many replicas a file will be available at. Each file replica has a version vector associated with it, which encodes the update history of the replica.

Since each file replica knows exactly where other replicas of itself can be found, it can let them know every time it is updated. The physical layer receiving an update notification waits until it can justify propagation costs and updates its local copies in a batch. Propagation utilizes an atomic single file commit service. For directories reconciliation is necessary. Events affecting the different copies of a directory are replayed to construct an updated version.

61.0.16 Volumes

Volumes are the basic structuring tool for separating disjoint subsets of all ficus files. Volumes are self-contained in that no directory references can cross volume boundaries. Files within the volume typically share replication characteristics such as replica location and the number of replicas. Volumes have unique identifiers. The volume identifier consists of the identifier of the host that created it and its own identifier within that host. The volume replica is specified by a replica identifier. Altogether, the volume replica is uniquely identified by the identifier of the host that created the volume, the volume identifier and the replica identifier. A graft point is similar to the traditional mount point. It specifies the root location of an entire volume. The graft point may contain any number of specific host identifiers and volume identifiers which can alternatively be used as a specific volume replica. Mounting of volumes at graft points is automatic.

62 How to Time-Stamp a Digital Document (Haber, Stuart and Stornetta, W. Scott)

[77]

This is one of the earliest published articles on the topic of secure, digital time-stamping. The task of digital time-stamping means assigning a time to a digital document so that it can be later verified that the document existed in exactly the stamped formed at the time stamped. To verify the validity of time stamp, all of the following must be provable:

- The document presented with a time-stamp is exactly the document on which the time-stamp was put
- The document was stamped with the time that was current when it was time-stamped

62.1 Naïve time-stamping scheme

The document is passed through a collision-free hash function. The resulting fingerprint is sent to a globally trusted Time-Stamping Service (TSS). The service signs the fingerprint with its signing key and returns it to the requester. The requester can check that the signature is valid, and that the signed fingerprint corresponds to the fingerprint sent. Furthermore, a challenger need only verify the signature locally – the TSS need not be contacted for verifications.

Issues with this scheme include reliance on TSS's clock, the assumed persistence of TSS's signing key, and the presumed honesty of the TSS.

62.2 A cornered Time-Stamping Service

A way to derive legal time-stamps from however dishonest a TSS is to use a *linking scheme*. The main idea is to link time-stamps together in such a way that later time-stamps contain some bits from earlier time-stamps. This forms a time ordering among documents on a time-stamp chain. In this case, the TSS is "cornered" because even if it wants to cheat, it cannot.

More specifically, a time stamp for a document fingerprint y_n requested by its owner ID_n at time t_n will be the following signed certificate:

$$C_n = (n, t_n, ID_n, y_n; L_n)$$

L_n is the linking information:

$$L_n = (t_{n-1}, ID_{n-1}, y_{n-1}, Hash(L_{n-1}))$$

When the $n+1$-th request has been processed, ID_n will also receive the identity of the owner of that request, ID_{n+1}. From C_n, the requestor ID_n can check that the time stamped on his document (t_n) is correct, that it's strictly higher than the time on the previous timestamp issued by the TSS (t_{n-1}).

To verify the timestamp, a verifier would first check the validity of C_n and ID_{n+1}. He then would contact ID_{n+1} to request his certificate C_{n+1}, ID_{n+2}. C_{n+1} contains a copy of y_n (in its linking section); it also contains a hash of L_n. If that checks out too, then all three, the TSS, ID_n and ID_{n+1} would have had to collude to deceive the verifier. The verifier can make the necessary number of colluders arbitrarily high, by continuing the verification process with ID_{n+1} and so on. Similarly, the verifier can proceed backwards to ID_{n-1} (which he can find in L_n) and request C_{n-1}, ID_n. In C_{n-1}, the verfier can find L_{n-1}, a hash of which is shown within C_n's linking section. Through linking, trust in the timestamp under scrutiny is not only strengthened by the existence of independent witnesses, but also by the discovery of different parts of a collision-free hash chain (that found within the linking sections).

The TSS would have to compute collisions for the hash function $Hash(.)$ to be able to embed a document within a hash chain. Furthermore, it couldn't back-date a document, at least not before the date of the preceding document in the chain, or the linking section would be invalid (i.e., it would be $t_{n-1} > t_n$). Finally, TSS couldn't forward-date a document, at least not after the date of the following document in the chain, since otherwise, it would be $t_{n+1} < t_n$.

The only way the TSS can cheat is by creating a long enough fake chain of time-stamps within which an illegally-stamped document can be inserted.

62.3 A widely witnessed time-stamping scheme

The general idea is to do away with a "trusted" Time-Stamping Service altogether. Instead, we rely on some of the other participants in a time stamping population to offer their view of what a time stamp for a document is.

More specifically, the fingerprint of the document we want to time-stamp is *stretched* into a list of identities using a secure pseudo-random number generator. This is list is just an interpretation of the generator's numerical result (for example, if the generator gives 320 bits, we turn these into 10 chunks for 32-bit numbers which we could interpret as 10 IPv4 addresses). The pseudo-random generator guarantees that we cannot "cook" its result so that it gives us a set of colluding identities given the fingerprint of a meaningful document.

Then, we contact all these identities with the fingerprint and ask them for a time-stamp. The signed results of all these identities form the time-stamp

for our document. As long as the number of participants in the scheme is large enough, or at least much larger than the probable number of colluders, it is highly improbable that we can get an illegal time-stamp out of this scheme.

Unfortunately, this scheme is not very practical. Random participants might not be highly available and random bits interpreted as identities might not correspond to existing participants. k-out-of-n schemes could be more useful in this context.

63 Subcontract: A Flexible Base for Distributed Programming (Graham Hamilton, Michael L. Powell and James G. Mitchell)

[78]

The gist is: Plug and play RPC mechanisms.

The general idea is that different objects elect what kind of RPC they need to operate according to their specifications (i.e., replicated RPC, atomic RPC, persistence or migration, etc.). This "kind" is embodied in a *subcontract* associated with an object. All RPC operations go through this subcontract.

Even in the cases where a different subcontract is expected, the mechanism supports subcontract on-line discovery; thus, if a client needs to operate on an object that supports a subcontract that hadn't been anticipated, the correct subcontract can be found and loaded at run time.

The basic operations supported by subcontracts are:

- marshal
- unmarshal
- invoke

The overhead over a more static approach is basically due to the additional indirect calls, every time an object operation is used (first to the object stubs and then to the subcontract). This is expense is justified, according to the authors, by the added flexibility and the enormous potential for extensibility, customization and optimization without the change of the base system.

64 The Zebra Striped Network File System (John H. Hartman, John K. Ousterhout)

[81]

The general idea is to do striping across multiple storage servers, with parity, just like RAID. Zebra globs writes together, just like LFS [148].

Striping helps with

- load balancing (load is distributed among all storage servers)
- parity for recovery (if a disk dies, the remaining ones can recover its contents)
- parity for availability (if a disk is down, the remaining ones can be used to recover the missing contents)
- higher throughput (aggregate bandwidth is higher than that of individual disks)

Striping is not done per file, which would limit the benefit (since the file would need to be chopped to tiny pieces), but per log segment. Per-file striping causes a discontinuity in the way small and large files are handled. If, on the other hand, small files are stored in fewer servers than the full array, then the space overhead of parity becomes huge.

The LFS approach was borrowed. Here, logging is used between a client and its fileservers (as opposed to the original LFS case, where logging was used between a client and its disk). Each filesystem client uses its own log and stripes its log segments, instead of the individual files.

File meta-data are managed centrally by a file manager. The logs contain only data blocks. The file manager also deals with cache coherence, namespace management. It has to be contacted for all opens and closes. It has a local "shadow" file for each file accessible through the Zebra filesystem. The local shadow file contains block pointers and such.

Segments are reclaimed by a stripe cleaner, similar to a segment cleaner in LFS. Instead of using segment summaries, as in LFS, the cleaner keeps track of usage information in stripe status files, which it keeps as normal ZebraFS files.

Deltas contain meta-information that clients write into their logs; deltas don't contain data. However, deltas are also read by the file manager, so that it can manage meta-data accordingly.

Recovery needs to cover:

- client crashes (in which case, a partially stored stripe might be regenerated from a parity fragment, or discarded)
- storage server crashes (in which case, either a fragment has been partially stored or some fragments should be stored but haven't; checksums help find incomplete fragments and parity helps reconstruct missed fragments)
- file manager crashes (in which case, metadata might not be up to speed given the current deltas; delta replay is used to bring the meta data up to speed, assisted by checkpoints of covered deltas per client).
- cleaner crashes (in which case, the cleaner status for each client's stripes might be out of date; log replays are used here as well).

The only critical resource is the file manager. Any single crash by another component can be dealt with on-line, without reducing availability. Recovery of any single failure can occur while the system is running.

Some points where things might not be as rosy:

- Small files are only collectively benefited. That is, if you're only dealing with few small files, you still have the problem with LFS not being the best paradigm to use.
- Striping introduces a higher error rate (more disks that can fail), which is then reduced back by parity striping units. It's unclear whether there's a net decrease in error rate.
- The larger the number of servers, the larger the stripe size has to be to make the fragment size efficient, i.e., more buffering is needed than in the single-host case. That's why stripe groups where introduced in the evolution of ZebraFS (see [8])

65 Application-Controlled Physical Memory using External Page-Cache Management (Kieran Harty and David R. Cheriton)

[82]

Exo-virtual memory.

The title says it all. Some applications need to play with their virtual memory. Examples are described in Stonebraker's rant in [163], in the case of databases. For another example, in the case of large scientific simulations, the application knows exactly how memory is going to be accessed (for instance, sequentially); therefore, it can best determine how page replacement is to proceed. Finally, applications can make educated decisions about space/time trade-offs, based on how much physical memory is available to them.

To accomplish that goal, the authors have modified the kernel of V++ to associate an external page-cache manager with each segment of a process's address space. This manager gets a number of pages from the system's manager and deals with them as it sees fit, to react to pages demanded from the associated application.

An interesting side note, explained for real elsewhere, mentions the allocation of memory according to an economic model of fixed supply. Application-controlled page-cache managers can only maintain memory as long as they can afford it. They get a steady income, get penalized for saving, and are subject to exchange rates with I/O traffic.

66 Recovery Management in Quicksilver (Roger Haskin, Yoni Malachi, Wayne Sawdon and Gregory Chan)

[83]

The key contribution of the system is the exposure of the low-level transaction mechanisms to servers, as opposed to the use of high-level transaction interfaces only. Therefore, servers can direct their own recovery, using these mechanisms.

Single-host transaction operations are handled locally. Only Transaction Managers need communicate with other hosts. Minimization of exchanged network messages was the motivating concern. The hierarchy is fault-tolerant and can be made more efficient, in that coordinators can be replicated (to avoid single-point failure), and coordinators can migrate, to minimize the distance between a coordinator and its participants.

Some refinements of two-phase commit protocols have been used. For instance, participants that do not modify recoverable resources, can be ex-

cluded from phase two for efficiency.

67 Monitors: An Operating System Structuring Concept (C. A. R. Hoare)

[84]

A straightforward paper, introducing an implementation of monitors (using the infamous "Hoare semantics").

Interesting here are some ideas on inserting some smarts into the mechanism that picks which waiter on a condition variable to wake up. The idea is to assign a fixed priority to a wait operation. This priority will be used to schedule wake ups when the condition variable is signaled.

A very useful set of rules Hoare describes in this concept was good enough that I'm going to quote them here:

1. Never seek to make an optimal decision; merely seek to avoid persistently pessimal decisions.

2. Do not seek to present the user with a virtual machine which is better than the actual hardware; merely seek to pass on the speed, size and flat unopinionated structure of a simple hardware design.

3. Use preemptive techniques in preference to non-preemptive ones where possible.

4. Use "grain of time" methods to secure independence of scheduling strategies.

5. Keep a low variance (as well as a low mean) on waiting times.

6. Avoid fixed priorities; instead, try to ensure that every program in the system makes reasonably steady progress. In particular, avoid indefinite overtaking.

7. Ensure that when demand for resources outstrips the supply, the behavior of the scheduler is satisfactory.

8. Make rules for the correct and sensible use of monitor calls, and assume that user programs will obey them. Any checking which is necessary

should be done not by a central shared monitor, but rather by an algorithm local to each process executing a user program.

68 Communicating Sequential Processes (C. A. R. Hoare)

[85]

Hoare presents a language where input and output and parallelism are basic primitives, and describes how a number of simple exercises would be accomplished on this language. He concludes by justifying some of his design choices.

The language consists of

- Guarded commands, i.e., commands with a condition and a body where the body is executed only when the condition executes successfully.
- Parallel commands, i.e., commands comprising multiple independent subcommands, all of which begin in parallel.
- Clearly named input and output commands. "Clearly" here means that both source and destination of a communication command are specific and unambiguous.
- Alternative commands, which are commands comprising multiple guarded commands; the guard most closely fulfilled is the one picked by the scheduler to result in the execution of a body.
- Repetitive commands, which are pretty much iterated alternative commands.

It is a cute and elegant enough language, with very powerful constructs. There is still some criticism though:

- *Explicit Naming* It's not fully justified, but it does make difficult the construction of libraries. The use of port names, to which processes can attach their inputs or outputs could mitigate the problem, without changing the semantics.

- *Unbounded process activation* In all process array examples, the bounds of the arrays are static. The reasons is that this is to run on a real machine, with a finite number of processors, so having a bound makes this language more practical.
- *Fairness* Exactly how a scheduler is going to pick that one command among many in an alternative command, which doesn't preclude progress is left open. It sounds like an interesting exercise to figure out how to do that, without building a very slow system.

A very enjoyable paper altogether, with some interesting insights on language construction in a rather different domain.

68.0.1 Exercise 4.6

Easy to do, but I thought I should...

```
S(i:1..100)::
  *[  S(i-1)?least()->S(0)!noneleft()
    []n:integer;S(i-1)?has(n)->S(0)!false
    []n:integer;S(i-1)?insert(n)->
      loaded:boolean;loaded:=true;
      *[  loaded;S(i-1)?least()->
            S(i+1)!least();
            [  l:integer;S(i+1)?l->S(i-1)!n;n:=l
             []S(i+1)?noneleft()->S(i-1)!n;loaded:=false
            ]
        []loaded;m:integer;S(i-1)?has(m)->
            [  m<=n->S(0)!(m=n)
             []m>n->S(i+1)!has(m)
            ]
        []loaded;m:integer;S(i-1)?insert(m)->
            [  m<n->S(i+n)!insert(n);n:=m
             []m=n->skip
             []m>n->S(i+n)!insert(m)
            ]
       ]
   ]
```

69 Querying the Internet with PIER (Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, Ion Stoica)

[87]

PIER is a project whose objective is to bring complex queries, similar to those possible over a relational database model, to loosely coupled, very large distributed systems.

The target application is aggregation of information over a large distributed set of data streams; for example, running queries over the tcpdump streams of many individual hosts spread around the Internet. When complex queries over such streams are necessary, doing the query processing where the streams naturally live, rather than shipping those streams to a central location for further processing, can drastically reduce network bandwidth and storage requirements.

PIER approaches the problem accepting that neither can consistency be strict in the results produced, given the scaling requirements and the loose coupling of the participating streams, nor can a standard schema be devised that might fit all interesting streams to aggregate.

PIER relies on a distributed hash table (DHT) over participating nodes for its operation. The DHT provides routing functionality, allowing nodes to join or leave the group of participants, as well as to locate the particular node that currently holds specific hash keys in the hash space. On top of a DHT, PIER places a storage manager that handles simple "put" and "get" requests for key/attribute pairs. The storage manager limits the lifetime of stored data; in other words, all data stored are "soft" and can only remain stored via periodic renewals.

PIER identifies data by a [namespaceID, resourceID, instanceID] tuple. A name space identifies a particular group of values; for the purposes of a relational data model, the namespaceID is the name of a relation. A resourceID corresponds to the primary key for a tuple. A namespaceID and a resourceID make up the hash key used within the DHT. Finally, the instanceID for a datum separates potentially multiple values (e.g., when a table is not indexed by its primary key).

Queries are defined as "boxes-and-arrows," i.e., by explicitly specifying

a query plan. Each operator in the plan takes its input from a queue and places its output in another queue; both queues are stored temporarily in the DHT.

Joins are implemented using a *symmetric hash join*. Every tuple from either participating relation — projected on the required attributes — is reinserted into a new, temporary table whose primary key is the join attribute; note that the temporary table will contain tuples from two different "shemata", one from each projected table participating in the join. In parallel, since joinable tuples from either table end up in the same node, as the DHT storage key is the join attribute, the local node receiving either tuple can fetch any potential matches locally, and then forward the resulting tuple to the consumer of the join (or the querier, if the join was the last operator in the query plan).

70 Efficient and Flexible Location Management Techniques for Wireless Communication Systems (Jan Jannink, Derek Lam, Jennifer Widom, Donald C. Cox, Narayanan Shivakumar)

[93]

[93] One interesting aspect here is that they are trying to motivate life long numbering. there is no actual motivation behind life long numbering; instead a viable way to obtain life long numbering is provided.

Life long numbering is provided by a hierarchical database scheme. Higher level databases store pointers to lower level databases. Leaf databases store the actual profile. Every database must contain a pointer towards all profiles included in its subtree. The top level database must contain pointers towards all existing profiles.

Queries start from the database of the caller going upward until a pointer to the callee is located (This happens for the first time in the database that is the least common ancestor of the caller's and the callee's databases); then the search proceeds downward.

Hiper is using this hierarchical architecture along with replication to both provide life long numbering and minimize query and update latencies.

The location of replicas is determined in a way similar to that described in[Shivakumar 94], modified to hierarchies instead of graphs. The number of replicas is determined from the available database space allocated to each of the systems users. Additionally, the more replicas there are in the system, the higher the update cost. Therefore, the number of replicas per person should be limited.

Replicas are placed in the hierarchy to minimize the update cost while minimizing the lookup cost. A useful metric for calculating those costs is one for calling frequencies between the sites of two people participating in a communication. This is called the local call to mobility ratio. It is defined as the number of calls from a user to a zone over the number of moves of that user in a fixed time period. When this number is high, it makes sense for the calling zone to replicate the profile locally. This ratio can be compared to a threshold to determine whether that user's profile will be replicated in the called zone.

Since ancestor databases contain the covered area of all their descendant databases, the local call to mobility ratio is always higher than the threshold in the ancestors when it is higher than the threshold in a leaf. So, naively, it would seem beneficial to replicate all profiles at as high level a database as possible. Unfortunately, higher level databases service more profiles than lower level databases which means that the update costs incurred could overwhelm them. A tradeoff has to be maintained between overwhelming higher level databases with updates and populating the hierarchy with ineffective profile replicas.

For high enough values of the local call to mobility ratio, replication can be considered justified. Similarly, for high enough values of the ratio, replication can be considered unjustifiable. For the values in between,replicas could be allocated in the most beneficient way, i.e., so that only those with highest ratio among the non- clearcut databases get a replica. The upper and lower threshold values are picked to unambiguously make replication good or bad.

The algorithm described in the paper first assigns replicas to as many of the databases with a ratio above the upper household as possible, starting from the bottom of the hierarchy, until all such databases are given a replica or the maximum number of replicas has been assigned. In the second phase, the rest of the unassigned replicas are allocated to databases with as high a ratio as possible, starting from the top.

Another contribution of this paper is a fairly elaborate framework for

simulating location management techniques. The simulator takes under consideration geographical features like freeways or local population sizes, when implementing user mobility and calling patterns, as well as network computing between sites.

The actual simulation setup in section 4 of the paper is a very interesting combination of statistical national averages with local counters in a simulation.

70.0.2 Issues

It is unclear why this algorithm was picked. Especially, it is unclear why exactly all of the second phase replicas are assigned top down, or why the split between bottom up and top down assignments is set to be the upper ratio threshold.

One big hole in the paper is the lack of proof of scalability of HiPER compared to other schemes. The authors admit that all their simulated systems are nothing special given current hardware and software.

71 Scenario-based performance analysis of routing protocols for mobile ad-hoc networks (Per Johansson, Tony Larsson, Nicklas Hedman, Bartosz Mielczarek and Mikael Degermark)

[94]

This paper picks up where reference[32] left off. It looks at two reactive routing protocols and one pro-active protocol, namely DSR and AODV on the reactive side, and DSDV on the pro-active side. Here the metrics studied are latency (delay), throughput, and routing overhead. The authors use the same random controlled scenario used in [32] (based on the random way point model), but also define a set of "realistic" scenarios: a conference scenario, a trade show scenario, and a disaster area relief scenario. Finally the authors introduce a mobility metric to help compare different scenarios with differing mobility patterns as to how mobile they are.

The mobility metric defined is based on the average absolute relative velocity of every node with respect to every other node. This is superior to the previous mobility metrics used, pause times between moves, since the latter

only applies to random way point patterns where every node uses exactly the same pause time as every other node and a constant speed during each move. Here a mobility measure between any two nodes is defined as averaging their absolute relative velocity over the duration of the scenario. Then the mobility metric of the scenario is calculated by averaging the mobility measures for all pairs of participating nodes. The authors have shown that this metric corresponds fairly closely to the rate at which links are established and broken and due to mobility. Unfortunately this metric does not account for oscillations, i.e. as in the case when an node moves back and forth relatively to another node.

The random scenarios simulated by the authors here are similar to the random scenarios simulated in [32]. Unfortunately protocol parameters were also arbitrarily chosen here. The different scenarios tried to isolate the effects of mobility and of increasing data rates.

Average delay was lowest with DSDV, since only packets with valid routes were delivered and longer routes rarely get established in the face of mobility. DSR did better than AODV due to its better caching properties. When data rates increased DSDV had more trouble converging. An interesting effect exhibited by AODV and DSR was a valley in latencies at medium mobility. At low mobility routes remained relatively stable resulting in longer queues at intermediate notes. At medium mobility as routes are in flux packets directed to the same destination get queued up at different intermediate nodes, creating some sort of load balancing, which is seen as lower delay.

Throughput wasn't affected much in the face of varied mobility by the pro-active protocols, whereas DSDV had serious trouble keeping up at higher mobility. For increased data rates DSR had more trouble than AODV because of the effect of source routes increasing the size of packets inordinately compared to AODV. Of course, that was more pronounced here than in a different scenario where packets are longer than 64 bytes.

Packet overhead was greatest for AODV because of its periodic liveness packets, whereas DSDV was constant with increasing mobility. Byte overhead was higher with DSR than with AODV since DSR piggybacks source routes with every packet. DSR showed a higher disadvantage for higher data rates. Again the packet size was a definitive factor here.

The three realistic scenarios described where arbitrary and limited in applicability, but went beyond what any other simulated scenarios had accomplished before. The conference scenario described the case where the speaker moves back and forth on the stage of a conference center, most attendees

remain stationary, and a few stragglers come in and out at low speeds. Data rates are low and sources are few.

That trade show scenario described cases where few well-connected clusters were forming around specific points in the area (booths) while a few nodes wander among booths joining or leaving the clusters.

The disaster areas scenario consisted of a few slow nodes in clusters which are far apart from each other and a few really fast nodes traveling among the clusters and interconnecting them representing vehicular tranceivers.

In most cases DSDV did well with latency but dropped quite a few packets. In the low mobility of the conference scenario DSR did better but in the medium mobility of the event coverage scenario AODV did better, because it could use its liveness periodic messages to attract mobility without an explicit route request. In the disaster area scenario DSDV did very poorly in terms of dropped packets since it didn't have enough time to form a longer route while the interconnections between clusters (the transceivers in the vehicles) where present.

Although the conclusions drawn about the protocols in both the random and the "realistic" scenarios are not broadly applicable except in the most general way, this paper is useful in that it pointed out the need for closer correspondence between reality and the scenarios used in evaluating ad hoc routing protocols.

72 SOS: Secure Overlay Services (A. Keromytis, V. Misra and D. Rubenstein)

[96]

This work addresses the problem of preserving the connectivity of a fixed, prearranged set of end-points on the Internet, in the face of aggressive, powerful denial of service attacks. The authors propose a virtual distributed firewall that is implemented as a secure overlay. The target application for such a service is the communication of a small set of principals who may be at different network locations, such as an emergency response team; such critical applications must be protected from DoS attacks that can curb their ability to communicate.

A number of high-powered, well-connected routers participate in the formation of a closed overlay that implements a DHT interface. Nodes in that

overlay may take the roles of *Secure Overlay Access Points* (SOAPs), *beacons*, and *secure servlets*.

A SOAP is an entity that authenticates end-hosts wishing to contact a target end-host via the SOS. Since all end-hosts protected by an SOS are known beforehand (at least to the operator of the SOS), a SOAP can independently authenticate such end-hosts and establish a security association with them for the duration of a session.

A secure servlet is a node in the overlay that serves as a last logical hop before a transmission is routed to the target end-host. The target selects a few secure servlets among those in the overlay (periodically), and allows their packets to pass through the physical routers or firewalls in its proximity. Filtering at those routers or firewalls is only done according to the source and destination addresses, to minimize the overhead; this is an important consideration, since during an attack, those routers will be under severe workloads.

A beacon is a rendezvous point within the overlay at which transmissions coming from a source end-host via a SOAP can locate the appropriate secure servlet so as to reach the target end-host. A target registers its current secure servlets with a specific set of beacons (located at specific DHT identifiers in the overlay). Source packets coming from a SOAP are directed to the same beacons and then forwarded on to one or more of the appropriate secure servlets.

To stop the transmissions from the source to the target, an attacker must eliminate (e.g., by flooding) all SOAPs that the source might use, all beacons associated with the target, and all secure servlets picked by the target. Ideally, any overlay node can play the role of a SOAP, so eliminating all of them can be a daunting task for an attacker. Beacons are placed at a DHT identifier, rather than at a particular physical machine; as a result, as beacons are taken down by an adversary, others take their place thanks to the self-healing properties of the underlying DHT. Finally, the target can create new secure servlets as current ones are attacked, or can remove a node from its set of secure servlets if it is found to be subverted, allowing attacking traffic to pass through (note that the IP addresses of the few secure servlets are the only secret an attacker needs to know to penetrate the physical firewall of the target).

The authors evaluate their design analytically, given a non-adaptive attacker, as well as a dynamic attacker who changes the nodes it is attacking as previous victims are removed from the overlay. An important consideration

in the analysis is how fast the adversary detects that attacking a current victim is no longer effective (because the victim has been removed from the overlay) and how fast the overlay can heal after an attacked node has been removed.

73 Disconnected Operation in the Coda File System (James J. Kistler and M. Satyanarayanan

[97]

Main goal: highly available operation for a network file system, even in the face of disconnection.

Workload is balanced at the granularity of subtrees. High availability is achieved through:

- replication
- disconnected use

Caching is done on whole files. This simplifies disconnected operation and allows for more straightforward failure modes. Clients are given most functionality, to promote scalability.

The contents of the cache, in the face of disconnection, can be determined by the user, in the same way this is done when a user copies interesting files on a laptop before leaving a network access point and then reintegrates them back into the file repository, on reconnection.

When disconnected, the pessimistic approach would be to enforce consistency, by requiring an a priori acquisition of a shared or exclusive lock for access. In the case of involuntary disconnection, this is unrealistic. The optimistic approach, which is the one taken by Coda, allows all updates and hopes no conflicts will ensue; when conflicts do arise, they are resolved upon reconnection.

System calls are intercepted through v-nodes and serviced by a user level cache daemon (Venus). Venus can be in any one of three states:

- When connected, Venus is hoarding (i.e., accumulating files in its cache). It has to deal with updating its file versions (since others might also be modifying cached files), with cache space management and with

keeping certain files, in which the user has expressed explicit interest, around.

- When disconnected, it emulates normal operation, by using its cache to satisfy file requests. It also fakes some of the functionality that normally only servers provide (i.e., assigning a new file ID and such). Results of this usurped functionality have to be revalidated by real servers upon reconnection. Cache misses can either block until reconnection, or return an error code. Mutating operations are logged, so that they can be replayed upon reconnection. A caching intricacy during disconnection is that the cache has to be maintained across crashes (since it's what the user perceives as the emulation of the servers). Therefore, it uses *recoverable virtual memory* for the meta-data, which enforces transaction semantics and provides recovery across failures.

- When reconnected, it briefly goes through the reintegration phase, where it deals with update conflicts. The entire log of mutating operations, as assembled during disconnection is shipped to all servers. There it is replayed inside a single transaction, and applied to all referenced files. If reintegration succeeds (i.e., there are no conflicts), the log is purged and Venus goes back to hoarding mode. Otherwise, a user utility can be used on the log to selectively apply it. Conflicts of interest are only write/write conflicts. Each file has a version number (storeID) associated with it. If the version numbers between the log entry and the server copy of the file are compatible, there is no conflict. In the case of directories, conflicts can sometimes be resolved.

74 Vnodes: An architecture for multiple file system types in Sun UNIX (Steven R. Kleiman)

[98]

The design goals for this extension were:

- to split FS-dependent from FS-independnt functionality
- to provide an interface between the two

- support popular FSes of the time (in terms of the offered functionality)

The implementation uses an object-oriented approach: operation vectors.

The Vnode is a virtual equivalent to an i-node. Vfs is a virtual mount entry. Different filesystems can appear in the same hierarchy without specialization.

75 A cluster-based approach for routing in dynamic networks (P. Krishna and N. H. Vaidya and M. Chatterjee and D. K. Pradhan)

[100]

Although generally adequate for relatively stable networks, distance vector and link state routing protocols are not good at dealing with dynamic networks, where topology changes drastically all the time. Current protocols place a heavy computational burden on the participants and have long convergence periods compared to the periods during which the topology is relatively stable.

The basic idea of this proposed approach is to divide the graph of participating network nodes into a number of overlapping clusters. A change in the network topology corresponds to a change in the cluster membership. A *clique* is a cluster where every member is reachable by any other member through a path of length 1. For example if all members of a cluster are within each other's broadcast range, then they form a clique. For paths of k hops over which members are mutually reachable, the concept of a k-cluster is defined. This paper deals only with cliques, not k-clusters where k is greater than one.

The goal sought is to find a "good" loop free routing between each pair of mobile hosts in the network, where the topological connectivity is subject to frequent, unpredictable changes. And good route from one host to another is comparable to the shortest route but not necessarily the shortest.

It is assumed that a node is aware of all its neighbors at all times, through a link level protocol function, all packets transmitted over a data link are received correctly in the proper sequence within a finite time, control messages

are processed one at a time. It is also assumed that the graph of all nodes is always connected .

Topological changes can occur because, first a mobile node turned on, second the mobile node turned off, third the mobile node came within range of another mobile node, and fourth a node went beyond range of another mobile node. At all times the network should remain *cluster-connected*. This means that every node should belong to at least one cluster, and, if the corresponding graph is connected, there is a path from each node to every other node through the edges of the clusters in the graph, requiring no hops connecting nodes that don't belong to the same cluster.

Each node maintains a list of its neighbors, the list of all clusters in the network, and the list of boundary nodes in the network.

Insertion of a new node The new node notifies all its neighbors of its arrival. Each of them sends the list of its neighbors and current list of all clusters to the new node. The new node recalculates the new clusters for its locality, making sure that no redundant clusters are included, and distributes the list of all clusters and all boundary nodes as it has been modified by its addition to its neighbors. These two new lists are distributed via all boundary nodes to the rest of the network.

Removal of a node One of the former neighbors of the removed note initiates a removal procedure. Only former neighbors of the removed node which used to belong to the same cluster as the removed node may initiate a removal procedure. Among those only one should initiate the procedure. That is the former cluster mate which used to belong to the largest number of common clusters with the removed node. The removal procedure aims to expand the clusters that shrank due to the removal. Multiple nodes might have to initiate their cluster expansion: for each cluster that the removed node used to belong to, a former cluster mate expands that cluster.

Nodes come within range of each other One of the two nodes, that with the highest identifier, performs the node insertion procedure.

Nodes go out of range of each other If the nodes belong to the same cluster, then one of them, that with the highest identifier, performs the node removal procedure (pretending the other node was turned off). The

node with the lower identifier just expands its local clusters. If the nodes did not belong to the same cluster then nothing happens.

After all current clusters and boundary nodes get distributed, each node constructs its routing table. This is done by running the shortest path algorithm on the cluster and boundary node information. Only boundary nodes really need to run the shortest path algorithm. Other nodes within the cluster can just send packets to one of the boundary nodes within their range.

76 VAXclusters: A Closely-Coupled Distributed System (Nancy P. Kronenberg, Henry M. Levy, William D. Strecker)

[101]

An intermediate construct, between tightly-coupled multiprocessors and loosely-coupled distributed systems. It's like a NetworkOfWorkstations approach, only for VAX machines.

The interconnect ports provide both a guaranteed, reliable, in order message service and a UDP-like, unreliable service. They also allow direct transfers of block data from virtual memory to virtual memory without the intervention of the operating system. Data is sent through the interconnect ports using 7 queues (4 outgoing data queues in 4 different priorities, 1 response queue for incoming data, and 2 pool queues with free messages and datagrams). The host program need only place the information it needs sent in those queues and the interconnect hardware will read the queue elements in the right priority and send them back, without bothering the CPU. Incoming data is placed in the queue and an interrupt handler can deal with all incoming packets in the queue, before returning, significantly reducing the delay through interrupts.

Disk systems (a disk with an interconnect component) can also participate in the network, providing storage facilities for all machines in the cluster. Access to those disks is also provided through the message-oriented interface described above.

Device Management is greatly simplified:

- single interface to all devices

- better sharing
- better usage

On each node, connection managers deal with locating resources, detecting recoverable failures, integrating new cluster members and so on. A quorum scheme is used to determine cluster membership. When operational members become fewer than the quorum value, then the cluster suspends activity until enough nodes have come back up. If a node fails and restarts before its membership times out, it's allowed to rejoin. Otherwise, it's told to reboot, before it can reenter the cluster.

Synchronization is handled by the distributed lock manager. This deals with awarding, maintaining and releasing locks of different types (shared, exclusive read-only, exclusive, ...) on objects of different types (from an entire file system to specific records of a file). For each hierarchical tree of objects, the master is the node that first requested a lock on the root of the tree. It has to keep track of granted locks and queues of waiters for incompatible locks. Lock masters are located through a lock directory, which associates a master with a resource name. Directories are also handled in a distributed manner. Given a resource name, a hash function yields the owner of the directory for that name (assuming a specific, well known number of members in the cluster).

During a cluster transition (e.g., addition of a new member, or escape of an old member), operations halt temporarily. All lock managers deallocate locks acquired on behalf of other systems. Then, each lock manager reacquires the remote locks it was holding in other systems, thereby redistributing masters and directories (since the membership number has changed).

All lock manager operations are optimized when applied locally.

This system was unique in that software and hardware were being designed together, with a specific common goal: the creation of a clustered system of simpler, commodity-like machines. When the hardware is given, many more compromises than listed here have to be made.

77 A Location Management Technique to Support Lifelong Numbering in Personal Communications Services (Derek Lam and Yingwei Cui and Donald C. Cox and Jennifer Widom)

[102]

[102] One obvious change from the previous paper ([93]) is that online, as opposed to offline, replication is investigated. In other words, replication decisions are made while the system is in operation.

A new thing in this paper is a reasonable albeit slightly fluffy method for maintaining the metrics used to calculate the local call to mobility ratio.

In this online version of the scheme (called *Hopper*), the threshold variables are replaced by a threshold variable for creating a new replica and a threshold variable under which a current replica is removed. The values of these thresholds can tolerate non-optimality, as the simulations show. The thresholds for replication and deletion are picked from the basic cost inequality, comparing update cost and lookup cost for any two pairs of zones. Values are picked by instantiating the hop distance between the caller and the callee with their minimum and maximum values respectively. In other words, what used to be the lower threshold in the offline case, becomes the delete threshold and what used to be the upper threshold in the offline case becomes the replicate threshold.

The scheme uses a time threshold to determine when calling or mobility measurements are too old to use. When the time threshold elapses, calling and mobility measurements are flushed. To make sure that early estimates after a metric flush are not too inaccurate, calling and mobility measurements are not used to calculate new ratios until their sum surpasses another predefined Threshold.

When after a lookup, it is determined that the caller deserves a replica, but there are no more replicas available to create there,then the replica exchange protocol is used. This is done asynchronously. During this exchange, a random replica is considered for deletion. If the randomly selected replica is not as beneficial as the caller's, then the former is deleted and the latter is notified that it currently contains a replica of the profile in question. The exchange is not performed if it's going to cause excessive transfer costs, for

example if it has to travel too far, or the system is currently overloaded. Replicas may also be deleted during updates. If when a replica is updated with a move, it determines that it no longer surpasses the deletion threshold, it notifies the previous current location of the profile that it has removed itself from the replica set.

Another departure from the previous, offline algorithm, is that replicas are now only placed into leaf databases. The authors justify this choice with its lower complexity, especially in the case of online replica allocation.

The simulation part of the paper presents a multi-day run involving the ten most populous cities in the USA. The basic home location registry technique (with translation for lifelong numbering), basic home location registry with caching, simple hierarchical, HiPER and Hopper are simulated. The simulations show that Hopper can achieve fairly consistent lookup locality, due to its online nature, while maintaining low requirements on database storage.

78 The Byzantine Generals Problem (Leslie Lamport, Robert Shostak and Marshall Pease)

[103]

The *Byzantine Generals Problem* is a theoretical, toy problem used extensively in Distributed Systems research. It is intended to model the case where a group needs to reach a common decision, in spite of the presence of saboteurs within the group.

The original definition of the problem follows:

Byzantine Generals Problem. A general must send an order to his $n-1$ lieutenants out in the field, such that

1. All loyal lieutenants obey the same order
2. If the general is loyal, then every loyal lieutenant obeys the order he sends

For the problem to be solved, the following two conditions must be true:

1. Any loyal lieutenant receives an order exactly as sent to him by the general.

2. If the general is loyal, then he sends the same order to every lieutenant.

If the messages sent by the general are not signed, then no solution will work if no more than two thirds of the participants are loyal.

79 Experience with Processes and Monitors in Mesa (Butler W. Lampson and David D. Redell)

[106]

This is an introduction to the monitor construct, as implemented on the Pilot workstation and in the Mesa programming language. Pilot was intended to be a multiprogrammed single-user machine. Therefore, a lot of protection-oriented issues are dismissed as only producible through malicious mechanisms.

The authors didn't want to solve communication with message passing, mainly because it didn't mesh well with their programming language of choice, Mesa. After all, Lauer and Needham argued that message passing and process-oriented structures are, effectively, duals of each other.

To guarantee consistency, they did not want to use a non-preemptive scheduling mechanism, mainly because the whole idea of paging (and the necessary page faults) breaks non-preemption, anyway, and besides, in that case, multiprogramming would have to be done only through voluntary relinquishments of the processor. This is, by common agreement, very awkward.

Processes in Mesa are very easy to create. In fact, any procedure can be invoked as a separate process; in other words, procedures can be called synchronously or asynchronously, with explicit support in the language. Process creation in that context (especially since it shares the address space and the program text) is very cheap, making arbitrarily frequent process creations a viable approach. The only difference between synchronous and asynchronous processes comes in the context of exception handling, since detached procedures are, in fact, the root in the exception handling pyramid. Perhaps exceptions could be passed on to the creator of a process, but that would probably be too big a hassle, worse even when the creator has itself terminated.

Monitors have entry procedures (which need to reserve the lock before they're executed), internal procedures (which can only be called from in-

side an entry or internal procedure, and don't touch the lock), and external procedures, which don't require the lock. Things get a bit hairy when an internal or entry procedure contains a call into a different monitor's procedures. The lock of the first monitor doesn't get freed, in which case, if the second monitor causes the caller to sleep, the first monitor becomes bound on a sleeping process. Alternatives exist to some extent, but it's a generally difficult situation, which makes the whole monitor concept less than perfect.

To make sharing of large, cellular structures more efficient, the monitor mechanism has to allow for finer locking (the default case deals at the whole Mesa module's scale). An idea is to have monitored records (such as a node in a linked list, for instance), allow locking at that scale. Obviously, when futzing with the structural integrity of the entire construct (front and back pointers, etc., or counters), perhaps the global lock should also be acquired.

Mesa monitors/condition variables define a different semantics (differing from C.A.R. Hoare's initial suggestion). A notify (which Hoare called "signal") doesn't relinquish control of the CPU immediately. It merely is a hint that any waiters on a condition variable might start thinking about waking up. When they do, they explicitly need to check the condition on which they slept, mainly because it might still not be favorable to their causes (e.g., a different process might have entered the monitor first, before the former waiter woke up). This may result in fewer context switches, though not necessarily in the face of congestion.

The authors have also defined an additional operation on a condition variable, called "broadcast". This wakes up all processes waiting on a condition variable. It can be invaluable in cases where a single condition change might satisfy the requirements of multiple waiters.

The authors define the concept of a naked notify. This is only performed by devices, via interrupt handlers. Since a device probably cannot go through the whole monitor entry story every time it needs to produce an interrupt (because others might be following), it just does an unprotected notify ("naked"). This can cause race conditions, since it might occur between the time that a real process has checked a condition and has decided to sleep. In effect, it might never be awakened again. A lock-free synchronization construct could be useful here (like compare-and-swap).

80 Designing a Global Name Service (Butler W. Lampson)

[104]

A great quote from the paper:

> "There are already enough names.
> One must know when to stop.
> Knowing when to stop averts trouble."
>
> Tao Te Ching

Context: Name service for all computers and all people in the world. In general, only very few pairs of principals will have any communication exchanges at all. Assume the names change slowly, and the properties associated with names also change slowly. The name service is also large, long lived, highly available, invulnerable to local faults and tolerant of mistrust. It all sounds a lot like a hierarchy.

From the client's standpoint, a hierarchical name tree is seen. Symbolic links are supported. The top part of the tree consists of directories. Each directory has a unique identifier (something like an i-node in a Unix File System hierarchy). From each super-directory, subdirectories can be reached through named arcs (file names in a Unix File System hierarchy). So, nodes in the tree are independent of the names through which they're reachable. From directory leaves, value trees spawn. Value trees are also name trees where an arc is a name. Nodes contain time stamps and a present/absent flag.

In the above scheme, we can name single values, sets of values, or subtrees. Each directory has an access control function that maps principals and name paths into rights (R, W, test). Each directory has an authentication function from an encryption key to a principal. Some mappings in those functions are maintained externally, since no Public Key Infrastructure is assumed (i.e., by a courier).

At the administrative level, replication and subtree/server associations are visible. Every tree directory arc, except for the directory identifier it points to, also contains a list of servers on which that directory identifier lives (any of them can be picked). Servers are also named in the name tree.

Updates originate at one of the copies and later on propagated to the rest of them. This happens during a *sweep* operation, which collects updates

from all copies of a directory and then spreads them through all of them, resetting their last sweep timestamp. The entire set of servers can't be reliably retrieved from the parent (after all, the parent might be undergoing update as we check it). So all servers are connected in a ring, through which a sweep proceeds to update all of them. Disconnections, network partitions, failures lead to disruption of the ring. Ring reformulations occur manually by an administrator, and they mean the advent of a new *epoch*.

Administration operations under different subtrees are independent and need not be coordinated. When, however, trees are merged together under a new root (i.e., name services are combined into a new one), a scheme has to map from the old well known root (for which an identifier, not a name, was known) to a node in the new tree. This is done in a table of well known nodes, which point from a directory identifier that used to be a root, to a path in the current scheme. For every such reorganization, the well known node table has to be translated to the new tree; this is done infrequently enough, that the extra computational overhead is not an issue.

Restructuring, that is moves of entire subtrees under another subtree, is handled using symbolic links, so that paths leading to the old location of a subtree have pointers to the new location of the same subtree.

To permit caching, each entry has an expiration time. No entry can change before its expiration time has passed (except, perhaps, it can move to a different subtree, as long as there's a symbolic link to its new location). Therefore, clients can safely cache copies of past lookups, until their expiration time.

Value updates must be:

- Total (we can't have them be undefined for a particular state of the database)
- Commutative (their order shouldn't count, or concurrency is going to be a pain)
- Idempotent (they should yield the same result, no matter how many times they're applied, to cover replays and other nasty little habits of distributed systems)

On this basis, updates are specified on timestamped paths (i.e., every component of the path has a timestamp identifying the particular node the client intended to follow). Earlier timestamps are superseded by later timestamps.

Directory lookups reflect the state of a directory after *all* updates prior to the sweep time of the directory, and some of the updates after that sweep time.

Every server stores a pointer to a server that's closer to the root than itself, unless the former server does store the root. Also, a server cannot contain the only copy of the directory that leads to it from the root. Otherwise, an infinite loop trying to locate that directory would ensue.

80.0.3 Issues

Network partitions have to have user intervention to be dealt with. That sounds kind of silly.

No mention of time skew. Times are assumed globally synchronized.

This pretty much describes how DNS works except for a few differences:

- Changes can be introduced to an entry, before it has globally expired.
- Entries don't have expiration times (since there's no reliable, global time), but time-to-live values.
- The root can never change, so instead of up pointers from servers, there is a well known table of root servers.
- Lookups are more permissive, since timestamps don't have to be a part of a query.

81 Authentication in Distributed Systems: Theory and Practice (Butler Lampson, Martín Abadi, Michael Burrows, Edward Wobber)

[105]

[105] This is an extremely dense paper describing a detailed theory for authentication and its practical applications.

Principals are the main acting entities in this authentication theory. Principals can request actions. Actions are performed on *objects* (resources such as files, devices, processes, etc.). The objective of this work is to provide a

language in which rules may be defined, according to which requests by a principal on an object are granted or not. Rules are attached to the objects they govern. A *reference monitor* decides whether to allow a specific action, taking under consideration the involved objects with their associated access rules, principals and requests.

The job of the reference monitor is non-trivial, especially in distributed systems, since it must make logical inferences leading it to a decision. For example, principals almost never directly attempt an action on an object themselves, since principals are humans and objects are bits; most usually, a principal utilizes a *channel*, such as a network connection, a pipe or a function call (e.g., to access object X, Alice, a principal, types on a keyboard, which is attached to a system K, connected through a TCP pipe P to another system L, where object X lives. The reference monitor on system L sees an action attempted by a process Z on object X.) Channels are principals, too; however, they're more than that, because they can request things directly inside a computer, whereas human principals can't.

The reference monitor must decide whether the immediate channel attempting an action is trusted at least as much as the principal mentioned in the access rule for an object. In the example above, if X specifies in a rule that it will accept all actions from Alice, then the reference monitor must decide whether process Z is at least as trustworthy as principal Alice. To do that, it has to reason about Z, L, P, K and Alice.

Trust can be bestowed or transfered by a principal on another principal. For example, I may trust my brother to do anything that I can do. To prove my trust, I give my brother a legally binding document, such as a power of attorney. This provable trust is represented in the authentication theory described in this paper by the "speaks for" relation. When A speaks for B, A can do at least as much as B can do (i.e., if a request should be granted for B, then it should also be granted for A). Since only channels can say things directly inside a computer, invariably human principals must have channels that speak for them, so that they can communicate their requests through a computer system.

In a distributed system, where many components from different devices conspire to see a request through to its destination, it is inevitable to place different levels of trust in the reliability or the loyalty of every single component. A well-designed authentication system relies for its security only on few trusted components, its *Trusted Computing Base*, or TCB; in other words, no component outside the TCB should be allowed to affect the security of

the system by misbehaving. Furthermore, the system is *fail-secure* when it is conservative in the face of failure: when it gives the wrong answer, it always denies a request it should have granted, never the other way around.

81.0.4 Theory and Concepts

To connect principals to other principals, define statements, etc., the following constructs are possible:

Speaks for $A \Rightarrow B$ means that if A does something, it is exactly as if B had done the same thing. $A \Rightarrow B$ is a statement.

Quoting $A|B$ means that A says that B does something. A is quoting B as doing or saying something. Keep in mind that A might be lying (so B might not in fact do what A quotes B as doing). $A|B$ is a *compound* principal.

Says $(A : s)$ is a statement, representing the fact that A says statement s, or A *is entitled to* claim s. A may or may not have uttered s, but we can proceed as though A actually has. In other words, as far as A is concerned, given his knowledge and capabilities, s holds true.

Aggregation $s \wedge t$ is a compound sentence, consisting of both s and t. Similarly $A \wedge B$ is a compound principal, consisting of both A and B.

Some basic, intuitive axioms or inference rules using the above constructs are defined in the paper, including modus ponens, distribution and monotonicity over $\Rightarrow$ of "says" over $\wedge$ and so on.

81.0.5 Handoff

One of the primary tools in this theory is the ability to express delegation of authority. A principal can utter a statement that allows another principal (most commonly, a channel) to speak for it. The main handoff rule is

$$(A : (B \Rightarrow A))implies(B \Rightarrow A)$$

The handoff rule can be expressed even more powerfully (without asserting more), by making the initiator of the handoff be a principal who just speaks for the owner of the handed off authority, i.e.,

$$((A \Rightarrow B) \wedge (A : (C \Rightarrow B)))implies(C \Rightarrow B)$$

81.0.6 Encrypted Channels

Encrypted channels are very powerful. They allow the untrusted, arbitrary handling of messages, without compromizing the contents' secrecy or integrity, through public key cryptography. Even if existing public-key cryptographic algorithms turn out to be insecure or too expensive, shared-key cryptography can be used to simulate public-key functionality. Therefore, even with a small TCB, a sane authentication model can be built using encryption.

Given that a computer node is trusted as a whole (including its operating system and administration), the first step to building secure end-to-end channels, for use in the authentication scheme described, is to build secure channels from node to node. A single node-to-node secure channel between any two nodes is sufficient, since the theory allows multiplexing over a single channel. The node-to-node scheme must allow rekeying (since encryption keys must change regularly).

The authors present the following mechanism to establish secure node-to-node channels (from A's view point; B does the symmetric). It is assumed that both A and B have their own node key-pair, which has been assigned to the node by a certification authority, and their own boot key pair, computed when the node boots up.

- A sends its own public boot key to B (K_a).
- A computes a random symmetric key (J_a) and sends it to B encrypted with B's public key K_b, i.e securely ($\{J_a\}^{K_b}$).
- A puts together the two symmetric keys (J_a, which it just made up, and J_b, which it received from B) and creates a new symmetric key K to use in its secure node-to-node channel with B. Instead of concatenating the two, it uses a one-way hash function on the concatenation, to avoid known-plaintext attacks. The new key is $K = h(J_a, J_b)$.

  Up to this point, A and B can communicate using K, up until they have to change keys, or one of them dies. However, to be able to talk about this channel, they have to name the key they're using (so that, for example, innocent or malicious replays may be identified). To create an identifier for this newly created key, they take the next step.
- A encrypts K with a secret, private master key K_{aM} and sends it to B. This is A's identifier for K. Together A's and B's identifiers for K

define this secure channel between A and B completely. Note that any arbitrary way of creating an identifier for K (with minimal conflicts) would also do.

After this exchange, A knows that $K \Rightarrow K_b$, i.e., everything that arrives encrypted by K can be safely trusted as much as anything that could arrive encrypted by K_b^{-1}. Since K_a and K_b are not used by the actual channel, they can remain in use for much longer (as long as the system stay up in this case) and still, the channel bears their trust.

To extend the chain of trust a little bit further, we need to show that someone "meaningful" sits behind K_a. That can be done by connecting the A's node key to K_a. Remember that K_{an} bears a certificate that $K_{an} \Rightarrow A$, for the "meaningful principal" A. Now, if node A can present something like $K_a \Rightarrow K_{an}$ (for example, sign K_a with K_{an}^{-1}), then K_a can be trusted as much as K_{an}, or transitively, A.

RIGHT BEFORE 5

82 On the Duality of Operating System Structures (Hugh C. Lauer and Roger M. Needham)

[108]

There are two schools of thought on building systems with synchronization: message oriented systems and procedure oriented systems. The authors propose a correspondence, element to element, of the two that renders them duals of each other, thereby making the question moot.

- message oriented systems:

  relatively static, large processes; specific communication paths; the process is the resource manager.

  Synchronization occurs at the message queues. Operations are serialized there.

  The canonical resource manager is a huge case statement.

- procedure oriented systems:

resources are protected by shared data structures; consistency is ensured by locks etc. Processes are short lived and numerous.

Synchronization occurs at mutex lock queues.

The canonical resource manager is a monitor.

A correspondence between elements of each class has been found. For instance, an asynchronous message-reply pair corresponds to forking out a process with a procedure call and waiting on the result.

The authors go even further, to suggest that the duality preserves performance as well. They break up performance in 3 components, all across which the duality is maintained:

1. execution times of the "meat" of each program

   unaffected by duality

2. overhead of primitives

   they assert it without proof; they seem to miss however that process creation is much costlier than procedure calls (perhaps in a threading context the differences are not as pronounced)

3. queueing times and scheduling

   scheduling should be mapped across in a straightforward way, since queues are involved in either model.

The conclusion is that the model or the performance considerations thereof shouldn't influence how a decision is made on which one to choose. Rather, considerations like which one is easier to implement could be used instead.

83 Selecting Cryptographic Key Sizes (Arjen K. Lenstra, Eric R. Verheul)

[109]

This paper describes a uniform, objective way to calculate the required cryptographic key size, given the assumptions and requirements for a particular application, such as longevity, potential cracking cost, etc.

A very important point repeated here is that key compromise is usually not the result of an inadequately selected key size, but of poor security practices. For example, if a highly secure private key is stored on a networked computer with poor security, protected by a simple-to-crack password, then the length of the private key won't make a difference - it need not be cracked; it can merely be picked from the computer that hosts it.

Along the same lines, if the security protocol that uses a very long key is flawed, the security of the key won't make the protocol good; an example given is that even if you have a very long password, if you treat the password in chunks when encrypting it, you reduce the overall security to that afforded by the size of the chunks, not of the entire password.

The authors make sure they point out that if practical quantum or molecular computers become feasible, not many of the current cryptosystems will survive, since they rely on the intractability of certain mathematical problems, given deterministic computers. They only present results of cryptosystems they consider unlikely to be proven useless in the current computational context, if current trends in technology and science continue.

They cover the following cryptosystem categories:

- Symmetric (e.g., DES with 56-bit keys, 3DES with 112-bit keys, IDEA with 128-bit keys, RC5 with variable key size and the Advanced Encryption Standard with 128-, 192- and 256-bit keys).
- Classic asymmetric (e.g., RSA, or traditional discrete logarithm systems such Diffie-Hellman and ElGamal)
- Subgroup discrete logarithm systems
- Elliptic curve systems

The authors briefly touch on cryptographic hash functions. Such functions are generally attacked in an exhaustive fashion, although they yield better results than symmetric cryptosystem attacks, since they are prone to the birthday paradox. The authors conclude that bit sizes for the result of a cryptographic hash functions follow the trends of a symmetric cryptosystem, of half the key size.

83.0.7 Symmetric Cryptosystems

The authors consider only exhaustive key search attacks for symmetric cryptosystems. No method that works better than exhaustive key search has

been identified.

The standard DES is considered insecure. A $130,000 machine was built in 1998, which can brute-force a DES 56-bit key in 112 hours.

Incomplete exhaustive searches increase the probability of finding an answer proportionally to the fraction of the space searched. Guessing can also be effective with symmetric cryptosystems.

83.0.8 Classic Assymmetric Cryptosystems

These rely on either the intractability of integer number factoring (RSA), or of the dscrete logarithm of integers in the multiplicative group of a finite field.

Exhaustive searches are not the most effective attacks on these cryptosystems. A fast algorithm for integer factorization, the Number Field Sieve (NFS), can attack RSA significantly faster. Furthermore, it can be modified to attack traditional discrete logarithm systems as well. For large enough keys, the attacks on the two subcategories are comparable in computation/storage requirements.

For NFS, it seems that special-purpose hardware cannot significantly increase cracking times.

Incomplete attacks using NFS are absolutely ineffective against these cryptosystems. RSA can be attacked incrementally by the Elliptic Curve Method, which can produce factors early with significantly higher probability than straight guesswork, although it is slower than NFS if it has to run to completion.

83.0.9 Subgroup Discrete Logarithm Cryptosystems

These are similar to the traditional discrete logarithm systems, but limit their operations within a subgroup of the multiplicative group of a finite field, where the size of the subgroup is prime.

Attacks against traditional discrete logarithm systems (i.e., NFS for the discrete logarithm) are also attacks for this category. Furthermore, other exponential-time, parallelizable attacks against these cryptosystems are also applicable (e.g., the *rho* method and the *baby-step-giant-step* method).

Guesswork is just as ineffective for these cryptosystems. However, one of the attacks specific to this category, the rho method, can be used incrementally, although it consistently has a lower success probability than fraction

of work done (probability is proportional to the square of the fraction of the work performed).

83.0.10 Elliptic Curve Cryptosystems

These are also discrete logarithm cryptosystems, with the difference that they use an elliptic curve group of a finite field (actually, a subgroup of such a group).

No subexponential attack against elliptic curve systems, similar to NFS, has been published. The best known attack is exponential in the size of the subgroup of the elliptic group. For key sizes around the maximum allowable key size in the Wassenaar arrangement, different tractability results have been published, ranging from the "reasonable" (for a $10 million machine and 32 days) to the "infeasible" for slightly more bits. The authors extrapolate between the two results in their calculations.

Guessing against elliptic curve cryptosystems is considered pretty much intractable. However, incomplete attacks using the rho method are, again, possibly effective.

83.0.11 The Model

The author's model for developing cryptographic key sizes depends on four factors:

Life Span Until when should the suggested key size be considered secure?

Security Margin How "secure" does the key have to be? This is harder to grasp, since there haven't been good ways to define "secure" in this context. The authors take a unique approach: DES is a well-known cryptosystem, whose performance is understood, and its demise was documented; in this work, the security margin is defined as the year until which we were willing to trust DES. For example, with the security margin setting at 1982, in 1982 we would consider DES to be adequately secure. A margin setting of 1984 is more permissive, since it was relatively easier to compromise a DES key in '84 than it was in '82.

Computing Environment How fast does the available computing power and memory double? "Available" here may have two distinct meanings. On one hand, it can mean "available, for the same amount of

money". Alternatively, this can also mean "feasible". The interpretation of "available" is flexible in the model described. The authors present results where the rate of doubling is 18 months (to mirror a popular interpretation of Moore's law) and "available" means "feasible", i.e., cost is irrelevant.

Cryptanalysis How fast does research halve the time it takes to attack a cryptosystem, all other things being equal? This is split into a cryptanalytic factor for classical assymetric systems and a factor for elliptic curve systems. The default settings expect improvements in cryptanalysis to halve attack times every 18 months for classical asymmetric cryptosystems, but not at all for elliptic curve systems.

83.0.12 Discussion

The authors proceed to describe the formulae capturing their model, as well as applications of the model to current key size recommendations and multiple verification rounds, given current data points.

This paper is a very useful general guideline for a reasonable forecast of what key sizes are adequate for a given life expectancy. Obviously, once the ballpark numbers are obtained, a more detailed, application-specific determination of the applicability of those numbers is necessary, taking under consideration performance requirements, for example. Most importantly, the guidelines given can be extremely useful in determining if the security design of a system is flawed: if we expect a high number of encryption operations with a very large key that is supposed to live for a long time, the numbers from this work might suggest an incompatibility between system requirements and practical expectations, which could lead to a redesign of the system.

84 Replication in the Harp File System (Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, Michael Williams)

[110]

Basic idea is replicating storage servers through a primary storage manager. Furthermore, writes are aggregated through the use of the write-behind strategy, in conjunction with a UPS, to make sure that power failures don't kill the volatile write log.

Harp is not a file system. Rather, it's a replication extension written using the VFS interface. It strives to maintain Unix and NFS file system semantics.

Every file is assigned a server group. A server group consists of a primary server, a bunch of backups and a bunch of "witnesses". When any of the primary or backups is disconnected, or partitioned away, a new view of the server group is created, where one of the witnesses is promoted to serve as backup. Witnesses only store event logs (i.e., sequences of modifications on the file system). Logs are played back during recovery to resynchronize the state of the file system.

Modifications on the file system, as stored in event logs, are applied by an applier kernel thread. This thread goes through the event log and calls the appropriate local filesystem calls to have them propagated to disk.

85 Building reliable, high-performance communication systems from components (Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth P. Birman, Robert L. Constable)

[111]

The basic thesis of this paper is that if we can decompose complex networking stacks into stacks of component implementations of "micro-protocols", small-enough fundamental functionalities that can be formally verified for correctness, then we can

- derive correctness proofs for the implementation of the entire stack, composing the correctness proofs of the components,
- rearrange the components into a different stack, and still be able to prove the correctness of the resulting implementation with minimal additional effort

- optimize each component, by extracting *bypass* code for the common case
- optimize entire stacks, by composing *bypass* code for the common cases of each component, reducing the latency inherent in layered systems, while still maintaining a proof for the correctness of the optimization

The authors combine expertise from the Ensemble project on composable component systems, and the NuPerl project on formal software verification to reach their goal.

Building systems out of components has many attractions:

- it is easier to design, develop, test and optimize small components of limited functionality than a large, complex system
- once adequately tested, components can be rearranged to form different systems, to adapt to changed needs
- components can be individually formally specified and verified, leading to a verification of the resulting system

However, component-based systems haven't taken over the world.

- Their modularized architecture imposes overheads stemming from additional function calls, poor locality of data along the control path, redundant code which is irrelevant in a particular configuration of the modules, and low optimization yields because of separate compilation of modules.
- Putting components together into a running system meant to meet a particular demand can be non-trivial, since each module may offer broad parametrization to promote reusability

The goal of this work is to produce stronger, provably correct communication systems, on one hand, while improving the efficiency of component-based systems.

This is accomplished as follows: First, the overall system functionality is decomposed into small, distinct *micro-protocol* components, each implementing a simple piece of the overall functionality, such as sliding window protocols, packet fragmentation, packet re-assembly, signing and encryption, etc. For each such micro-protocol, an *abstract behavioral specification* is

constructed. This is a description of how the micro-protocol is supposed to behave, how its state is supposed to change in the face of external events, etc. This abstract behavior is used to derive a formal proof of a set of properties for the protocol, e.g., whether it implements a first-in-first-out policy.

At the same time, to follow the abstract behavior (*what* the protocol should do), a *concrete behavioral specification* is designed, describing *how* the protocol is to attain the abstract behavior. From this concrete specification, an actual implementation can be derived (usually, through an automated method).

Through these steps, there is a formal proof that a certain micro-protocol has a given set of properties, and an accompanying implementation of that micro-protocol.

A stack of such micro-protocols can be optimized by creating *bypass* code for a set of hand-picked common cases, e.g., in IP, the case where a packet has a large enough TTL, doesn't need fragmentation, and has correct headers. This code is produced by reasoning about the concrete behavioral specification of each micro-protocol, following the right branch of a conditional command, and compacting the branches into a control path specific to the common case handled. Using the transformations possible on the language of the concrete specification, the bypass code can be proved equivalent to the stack of component layers under the common case applicable. This is done by first producing bypass code (under the common cases) through each of the micro-protocol modules, and then composing the associated correctness proofs to prove the correctness of the bypass code for the entire stack.

Although the technique seems highly flexible, it is still very time-consuming to produce the specifications for each micro-protocol. The basic premise that a few simple yet reusable micro-protocols can be produced to cover a large enough set of complex functionalities has not been shown one way or another. Assuming that such, highly reusable micro-protocols can be specified and formally proved, the technique could have wide applicability.

Unfortunately, only different types of the application of the described techniques have been compared to each other for performance. From this paper, no direct comparison to other similar, non component-based systems can be drawn.

86 An Attack on the Needham-Schroeder Public-Key Authentication Protocol (Gavin Lowe)

[112]

The author presents a way in which the public-key authentication protocol presented in [127] can be subverted with a man-in-the-middle attack. The general idea is that A starts a conversation with the intruder. The intruder can make someone else (B in the example below) think he's talking to A.

1. $A \rightarrow I : \{N_A, A\}^I$

   A starts a normal authentication conversation with I, the bad guy.

2. $I(A) \rightarrow B : \{N_A, A\}^B$

   I decrypts the message by A, since it was addressed to it. Then, I impersonating A, replays the message, encrypted with B's public key.

3. $B \rightarrow I(A) : \{N_A, N_B\}^A$

   B naively responds with its own nonce. The message is intercepted by I.

4. $I \rightarrow A : \{N_A, N_B\}^A$

   I just replays the message as received to A.

5. $A \rightarrow I : \{N_B\}^I$

   A replies as normal, exposing the nonce to I.

6. $I(A) \rightarrow B : \{N_B\}^B$

   Now I resends the nonce encrypted with B's public key. For all intents and purposes, B thinks it's talking to A now, although A thinks it's talking to I.

The protocol can be fixed against this attack, by including the identity of B in the middle step of the protocol, so that A cannot be tricked into colluding with I.

1. $A \rightarrow B : \{I_A, A\}^{PKB}$
2. $B \rightarrow A : \{B, I_A, I_B\}^{PKA}$
3. $A \rightarrow B : \{I_B\}^{PKB}$

87 How to Build a Trusted Database System on Untrusted Storage (Umesh Maheshwari and Radek Vingralek and William Shapiro)

[113]

The problem that the authors solve in this paper is that of maintaining trust in a large database through the use of only limited trusted storage. The database contents must be protected against unauthorized read and write access.

The complete trusted database produced in this work offers a long list of features: atomic updates, incremental backups, (low) concurrency, high scalability, object friendliness, and data indices. Features are modular, and they can be removed at will, to reduce the footprint of the database.

88 Threads and Input/Output in the Synthesis Kernel (Henry Massalin and Calton Pu)

[114]

The main ideas: code synthesis and reduced synchronization inside the system.

The principle of *frugality* says that "we should use the least powerful solution to a given problem".

Code synthesis is used to specialize and optimize system interfaces for their intended use. Especially frequently traversed interfaces (like the system calls interface) can save a lot of computation being specialized and made short enough for their associated thread. The basic disadvantage is that every thread, or every thread group, needs to store its special versions of common code, after its creation.

Reduced synchronization shares some of its motivation from specialization, above. The idea is that, instead of using data structures that support synchronization, even when synchronization is not needed, a non-synchronized version of them is created and used when the extra functionality is not

needed. Other ways of reducing synchronization is code isolation, procedure chaining and optimistic synchronization.

Code isolation transforms algorithms so that synchronization is unnecessary. For instance, in a regular data structure supporting a `put` and a `get`, a different piece of the meta-data in the structure is writable by the `put` and a different one is writable by the `get`. So locking is not necessary. Obviously, this works if there's only one process running `get` at a time, and only one process running `put` at a time, which is also where specialization is applied.

Procedure Chaining replaces context switches among threads in the same address space with simple procedure calls. This is done by changing the return addresses in the stack. *Optimistic Synchronization* can be seen, for example, in context switching, when a thread that doesn't use the floating point facilities of the processor does not save floating point registers when scheduled out. Floating point instructions cause an illegal instruction fault when first used, at which point the context switch code is make less optimistic, i.e., starts saving floating point registers as well.

89 A toolkit for user-level file systems (David Mazières)

[115]

This paper describes the design and evaluation of a toolkit that aids in the construction and debugging of user-level file systems accessible via an NFS interface.

The motivation behind this work is that file system development is very difficult: in-kernel file system interfaces vary significantly from OS to OS, making it hard to write and port across multiple platforms a new file system. Furthermore, non-POSIX access to file system facilities can be very important for some applications: it allows asynchronous, non-blocking file operations (beyond the asynchronous read/write covered by POSIX), and masks out certain security race conditions that otherwise would be deemed too complicated to be addressed.

Loopback servers had been used traditionally for automatic file system mounting. The automount problem was solved with difficulty before the toolkit provided in this work, because the design of NFS servers first made in-place mounting of remote file systems impractical, and, second followed

a monolithic locking discipline resulting in long delays. Other approaches using loopback servers, including the CFS cryptographic file system, the Alex ftp file system, and the Byzantine-fault-tolerant file system [40] were complicated, in some cases deadlock-prone, and required a lot of effort for otherwise common functionality.

File systems have also been developed using intercepted system calls or shared library replacements. In either case, practical issues, such as use with statically linked programs, for example, limited the applicability of these approaches.

Other promising approaches to FS development include stackable file systems, which are more efficient that loopback NFS servers since they are implemented inside the kernel but require a VFS kernel layer, and using kernel functionality from user space via special, non-standard device drivers, which have to be installed in advance.

To access a FS implemented as a loopback NFS server, a client uses a standard, in-kernel NFS client that communicates with a local agent posing as an NFS server. This local agent disguising as an NFS server implements the actual file system functionality.

NFS loopback servers have several shortcomings that the SFS toolkit addresses:

- Implementation of a FS in user space results in more context switches. The author argues that the extended functionality and rapid development of NFS loopback FSes can, in most cases, outweigh the performance hit.

- A loopback server is expected to multitask: it cannot ever use blocking operations. This is necessary since, as a user application, the server is open to blocking imposed by the kernel. However, since the kernel might require services offered by the loopback server before it can proceed with other blocked requests, essentially a blocked loopback server can cause a kernel deadlock. This means that a loopback server cannot use block functionality, not even via multithreading.

- Loopback servers must deal with the peculiarities of different standard implementations of NFS clients. For example, an NFS request that does not receive a response quickly may retransmit and lock its local mount point. An NFS loopback server must be able to deal with this and other characteristics of standard NFS implementations.

- NFS client implementations take "unspoken" liberties with the NFS protocol; for example, blocks transported to an NFS client from different devices do not carry different device identifiers in the associated field. Loopback servers must deal with such semantic issues. One remedy is to use multiple mount points, one for each semantically different file collection, although multiple mount points have other problems, as seen below.

- Finally, loopback servers, although less likely to crash the kernel, can still end up rendering a machine unusable through resource attrition: by holding file descriptors, mount table entries, etc. and then crashing.

The SFS file system toolkit relies on the *nfsmounter* daemon for its good relations with the hosting kernel. The nfsmounter is responsible for housekeeping duties where loopback servers operation. It remembers enough information about all running loopback servers, so as to be able to unmount a file system whose server has crashed. Specifically, if the daemon detects a crashed loopback server, it takes over the role of that server, returning errors for most requests, and helping the in-kernel NFS client to unmount the crashed file system. The nfsmounter runs using a reserver user and group identifier, so as to allow loopback servers to react differently to normal requests from how they react to nfsmounter requests.

This "special" status of the nfsmounter helps, for example, build a better automounter. The SFS automounter always responds immediately to a mount request with a symbolic link to a special local directory. Subsequent requests from clients to retrieve the target of the symbolic link are held up, *without locking the common mount directory of the automounter*. However, subsequent requests from the nfsmounter, which attempts to actually mount on the newly created directory an actual remote or local file system, are directed to the actual mount point allocated by the automounter. In this way, multiple loopback file systems can be mounted by the same automounter, without causing excessive delays in the case of disconnected networked file systems, for example.

To deal with the requirement for asynchronous operations within a loopback server, the author has written an asynchronous C++ I/O library, that allows an application to receive callbacks for I/O, termination of child processes, signals and alarms.

To avoid the need for large contexts within a callback, the author has created a macro that bundles arguments to a callback that are known in

advance into a callback object. For example, a callback that would look like `result function(a, b, c)`, where `a, b` are known in advance, before the callback is scheduled, can be bundled into `result callback(c)`, which implicitly calls `function` with the original two arguments, and whatever the callback is called with when it is issued. This is a crude implementation of a continuation in C++.

To deal with release of allocated resources, the author has used a reference counting scheme and garbage collection.

The toolkit allows the creation of *manipulations*, stackable transformers that can be attached to a loopback server, offering functionality that is independent of the particular server implementation. Such manipulators handle, for example, kernel bug workarounds, multiplexing of streams, etc.

The toolkit also supports the development of NFS loopback *clients*. Such clients allow an application to access a local file system via an NFS interface, as opposed to the normal POSIX file I/O interface. This can increase performance, in some cases, since an application can now overlap disk requests, which yields better disk scheduling. Furthermore, a root NFS loopback client can still use normal access controls without having to reimplement them. Finally, the number of open files limitation in most systems can be circumvented thus. Of course, the fact that loopback clients must run as root makes them sensitive, especially since NFS file handles can be abused to access any information on the target file system.

To demonstrate the usefulness of the toolkit, the authors has produced two sample file systems built using loopback facilities, *dumbfs*, which is a repeater of normal NFS, and *cryptfs*, a cryptographic file system. Developing these systems was very easy, with very few lines of code in addition to the actual toolkit. Specifically, the cryptographic file system duplicated functionality that CFS had to implement from scratch, without reusability.

All in all, this work is very exciting, and has opened many avenues for further research on file systems.

90 A Fast File System for UNIX (McKusick, Marshall K. and Joy, William N. and Leffler, Samuel J. and Fabry, Robert S.)

[116]

Problems with the "traditional" UNIX file system:

- i-nodes are separated from their data.
- blocks are allocated randomly, sometimes resulting in long seeks per 512-byte transfer.
- free blocks were organized in a linked list. The list's locality quality deteriorated after use.

First wave of changes:

- increase of block size to 1024
- fsck for maintaining reliability

The real FFS had the following enhancements

- superblock replicated at well-known locations. Each replica is located on a different platter, cylinder or track.
- another increase in block size: 4096k
- disk is divided in cylinder groups, each containing its own i-nodes, datablocks, block map etc. This ensures that access locality can be maintained to some extent. It also means that some free space needs to be reserved to allow this to happen.
- access locality is also maintained through optimal rotational placement of consecutive blocks on disk. The characteristics of the disk are part of the info in the superblock and the cylinder summary info.
- allocation policies (local with full info/global with partial info) are used to decide where an i-node or data block will go.
- blocks are split into fragments to minimize waste through fragmentation. Fragments can only be used at the end of a file. This means that appending into files with trailing fragments might result in the need to copy those fragments into new blocks.
- instead of a free list, a block map is used. In fact, this is a fragment map.

Additional Functionality, which wasn't coming as a direct response to shortcomings in the traditional Unix File System

- long file names
- advisory locks.
- symlinks
- "atomic" rename
- quotas

91 Development of the Domain Name System (Paul V. Mockapetris and Kevin J. Dunlap)

[119]

How the Domain Name System was born.

In the begining, there was the `HOSTS.TXT` file. This was a file whose lines contained an association between a name and a network address. This file was centrally maintained and distributed to all hosts on the network periodically. This worked fine, while there was a single, humongous host per organization; when, however, host numbers started paralleling user numbers, `HOSTS.TXT` became a huge administrative bottleneck: centralization was incompatible with the higher rates of change.

The main constraints that the DNS design strived to uphold were

- to provide at least as much functionality as the old method
- to permit distributed administration of the database
- to make name extensible and practically limitless

To deal with the distribution requirements, a hierarchical name space was chosen. Since this was a project intended for immediate deployment, trade-offs between the lowest common denominator approach and the featurerama approach were biased towards the former. A leaner protocol would have fewer points on which objections could be based.

The main components are the *name servers* and the *resolvers*. The former maintain the information and answer queries. The latter live on the client side and talk to client programs; they run the algorithms necessary to locate the name server with the sought information. The introduction of separate resolvers in the architecture was intended to facilitate use of the protocol by clients with very limited resources. Such clients could share a resolver.

92 The Click modular router (Morris, Robert and Kohler, Eddie and Jannotti, John and Kaashoek, M. Frans)

[120]

This is a very well-written, solid paper on the design, implementation and evaluation of a router built out of reconfigurable software components.

The basic idea is that most router functionality can be abstracted into a functional packet-flow graph, where vertices are primitive operations running on packets, such as checking the syntax of the packet header or decrementing the TTL of an IP packet, and edges are paths over which packets flow. The authors have designed a system where such a graph can be described using a high-level language, and then compiled into a downloadable Linux kernel module, which can run at speeds comparable to a monolithic implementation of the same aggregate functionality. The authors have demonstrated a simple IP router which comes within 10% of the performance of the monolithic IP routing module written for the 2.2.10 Linux kernel.

The basic building block is an *element*, which has a number of input and output ports, a particular packet processing functionality, and an optional configuration string that makes the element specialize its operation.

Elements are distinguished between two basic categories: *push* and *pull* elements. Push elements are put into action when they receive packets at their inputs. They process their inputs immediately and push them out to the elements to which their outputs are connected (via graph edges). Pull elements, on the other hand, are put into action when they receive a request on their outputs for a packet, i.e., when the elements to which their outputs are connected pull for a packet. When that happens, a pull element pulls in turn a packet from the elements to which its inputs are connected, and once it receives a packet, it processes it and returns it to the elements that

woke it up in the first place. An example push element would be the network interface device driver, which is awakened when it receives a packet on the wire, it processes that packet and then pushes it to the next element in line. An example pull element would be a packet scheduler, which is asked for the next packet, at which time it has to figure out where to obtain (pull) a packet from, before it returns it to the element requesting it.

In general, push elements can only be connected to pull elements, and push elements can only be connected to push elements. Queues, which accept pushed packets into their inputs, buffer them, and return them to their outputs when they receive a pull, provide the connective elements between push and pull. The authors also describe a number of other rules which can filter out incorrect graph configurations. For example, a single push output cannot be connected to more than one other elements.

Some elements need information that is not available to them within a packet they receive, to perform their tasks. For example, a *Random Early Drop* element needs to poll the lengths of queues downstream from it, before it can decide whether to drop or not a packet. Control links to provide such information can be established at compile time (i.e., when the element graph is turned into a runnable module). An element that needs this kind of information is said to use a *flow-based router context* with a particular filter (e.g., "stop searching for downstream elements when you find a queue"). The flow-based router context algorithm used here is resistant to cycles in the graph, and has a deterministic outcome.

A number of different types of IP routers are described in terms of the architecture in the paper, showing how elements can be reused, and reconfigured at runtime on a running machine. Some of the limitations of the architecture include the absence of type checking of element connections, its inability to describe modularly elements whose control flow does not follow packet flow, and its rather simplistic CPU scheduling algorithm.

From the performance evaluation, it looks like the number of elements in the critical path of a packet can increase latency quite a bit. However, this cost can be offset by the benefit of being able to change the fundamental internal configuration of the router, without the need for low-level changes. For low-end router applications, the benefit might be justified by the cost.

93 An Overview of SNIF: A Tool for Surveying Network Information Flow (J. Alves-Foss)

[121]

This paper describes ASAX, a rule-based audit-trail analysis system. ASAX features a language for expressing monitoring/alarm tasks that loosely resemble specifications of finite-state machines. Individual clients within a LAN run a logger that collects security events, such as a failed login. At each client, rules are installed that can perform host-level analysis of logs, correlating events from different applications. In parallel, logs are transformed into a standard log format, and then forwarded to a central monitoring station for the entire LAN, which then performs similarly specified network-level analysis of the collected audit trails.

94 A Low-Bandwidth Network File System (Athicha Muthitacharoen, Benjie Chen and David Mazières)

[123]

This paper revisits the problem of building an efficient, responsive network file system that can be used productively over wide-area network distances. Current solutions, such as AFS, Coda, NFS or CIFS operate well when network delay and bandwidth are abundant, but degrade fast and by a lot in unfavorable network conditions. The Low-Bandwidth Network File System builds an NFS-like file system with significantly reduced requirements from the network connection between the client and the server, at a processing and local storage cost.

The main idea is to replace the concept of disk blocks with that of *byte chunks*. A chunk is a sequence of file bytes, like blocks. However, chunks have variable sizes around an expected size. Chunk boundaries are determined by file content. Every overlapping 48-byte region of the file is hashed; if such a region results in a hash of particular pattern (a fixed representation of a fraction of all hashes), then that region is considered a chunk boundary. Chunks also have a minimum and maximum size, to compensate for "bad"

byte sequences that either yield no boundaries, or too many boundaries. The chunking algorithm is well-defined, and deterministic, and can therefore be computed on a file's contents independently with the same result.

All file chunks in a file system are indexed by their fingerprints, i.e., the result of a cryptographic hash function on their contents (fingerprints are almost certainly collision-free). Read/write operations between the client and the server are performed first using chunk fingerprints. If the communicated chunk fingerprint is found in a fingerprint database, then the actual chunk's contents need not be transferred over the network.

For example, when the client wishes to read a file, it asks the server for the chunk fingerprints for that file. A round-trip later, the client can pick and choose which chunks to request from the server, so as not to bring chunks it already has. The client might have some chunks since, for certain file work loads, contents tend to exhibit a lot of commonality. Similarly, when writing a file back to the server, the client sends all the chunk fingerprints of the file, and only transmits full chunks if the server doesn't find chunks by the same fingerprints locally.

Another helpful characteristic of content-based chunking is that inserting data in the middle of a file only affects at most two chunks of the original file, whereas in a fixed-position blocking scheme, all blocks would change (shifted by the size of the new data).

Shift-tolerance, the use of chunk fingerprints, as well as use of compression, make LBFS perform more than an order of magnitude better than traditional network file systems over common interactive and non-interactive workloads.

The cost of LBFS lies in two operations: first, the determination of chunk boundaries for every file-system file; second, the calculation of expensive, cryptographic hashes over entire file contents; and third, the indexing and maintenance of fingerprint databases. The authors found that given current processor and storage capacities available to mobile computers, these overheads are certainly justified.

95 Certificate Revocation and Certificate Update (Moni Naor and Kobbi Nissim)

[125]

This work extends previous techniques for the maintenance and distribution of certificate revocation information so as to reduce communication and computation costs. Starting from the basic model of Certificate Revocation Trees (published in [99]), the idea is to allow hash trees to change efficiently, as opposed to rebuilding the tree every time its contents change.

The authors use 2-3 trees, a balanced search tree.

One interesting point that is mentioned but not explained in the paper, with regards to general hash-trees, is that the data source, the Certification Authority in this setting, not only has to sign the root of the hash tree, but also the height of the hash tree. Presumably this precludes a certain kind of attacks against the collision-resistant hash function used, although it is not clear exactly what that buys.

The paper gives a very clear theoretical comparison of Certificate Revocation Lists, Micali's Certificate Revocation System, and the proposed updatable Certificate Revocation Trees, based on primary variables such as the total number of certificates, average concurrently valid certificates, fraction of certificates that are revoked before they expire, etc. Although simplistic, this comparison method gives some useful insight into how the different schemes compare to each other. Specifically, CA-to-directory traffic (i.e., from the data source to the untrusted data dissemination nodes) is greatly reduced with the scheme proposed in this paper. However, query traffic among end-clients verifying the validity of certificates and the directory is increased from the Certificate Revocation System to the proposed scheme (because validity proofs offered by the directory are larger — logarithmic in the number of all certificates — than those in CRS). Still, because of the heavier computation involved during CA-to-directory exchanges, CRS cannot scale to faster updates, but the proposed scheme does.

96 Using Encryption for Authentication in Large Networks of Computers (Roger M. Needham and Michael D. Schroeder)

[127]

Seminal paper on the basic authentication protocols used today. It covers both shared key ciphers and public key ciphers.

They're dealing with encryption and authentication protocols, in three

contexts:

- Interactive encrypted communication
- One-way encrypted communication
- Signed communication

In all cases, there are two phases: one during which principals request a key from a trusted authentication server and one during which they authenticate themselves with each other. Nonces (one-use-only identifiers) are embedded in most messages that might be prone to replay attacks.

The basic, shared key, Needham-Schroeder protocol between two principals, A and B, and a trusted authentication server AS are as follows:

1. $A \rightarrow AS : A, B, I_{A1}$

   I_{A1} is to avoid having a bad person send us an older response by the AS, which it (the bad person) has had time to decrypt and play with.

2. $AS \rightarrow A : \{I_{A1}, B, CK, \{CK, A\}^{KB}\}^{KA}$

   Both I_{A1} and B are returned, to make sure this is the response to the request above, and not just any replayed response.

3. $A \rightarrow B : \{CK, A\}^{KB}$

4. $B \rightarrow A : \{I_B\}^{CK}$

   This is to certify that the real A is on the other side, not just a replayed stream.

5. $A \rightarrow B : \{I_B - 1\}^{CK}$

   Obviously, this couldn't be $\{I_B\}^{CK}$, since that could just as easily be replayed...

The same scheme is followed for public/secret key encryption algorithms. Given that A and B have already located the public keys of each other, here's how they initiate a conversation.

1. $A \rightarrow B : \{I_A, A\}^{PKB}$
2. $B \rightarrow A : \{I_A, I_B\}^{PKA}$

3. $A \rightarrow B : \{I_B\}^{PKB}$

Unfortunately, the protocol above has been found to be broken. Specifically, in the end, right after step 3, B thinks he's definitely talking to A, although there's no guarantee of that. Gavin Lowe presented the first mention of the problem in [112].

97 Denial of Service (Roger M. Needham)

[126]

In this short paper, Needham focuses on denial of service as the most important attack against secure systems. He approaches this type of attacks through an alarm system.

Denial of service attacks that target the network can be either suppression attacks (preventing the system from offering its service to some or all of its clients), frivolous message attacks (simulating particular behaviors of clients), and flooding attacks (that aim to slow down or disable the network). Attacks on system components themselves can range from destruction to substitution (physical machine substitution or logical subversion).

Of particular concern are attacks that can bring a security system into disrepute. Burglars can cut the wires of the alarm system and then leave; the police realize that the alarm system is faulty and schedule a replacement but, in the meantime, the burglars come back and finish the job. However, such attacks are only feasible when they can be targeted; if the entire system appears faulty, swifter reaction is more likely. Therefore, a viable defense is to prevent the attacker from mounting directed attacks, thereby making it hard for him to avoid detection. Messages, for example, should be as unidentifiable as possible, to prevent directed message suppression.

Flooding is also an important attack, whose only defense is being able to discard material arriving at line speeds while also performing normal operations.

The author suggests that to prevent undetected attacks, the defenders must entangle a continuous service with the resources of the service being protected. The only way to attack the protected service should be passing through the continuous service whose problems are more frequently checked.

98 A Uniform Name Service for Spring's UNIX Environment (Michael N. Nelson and Sanjay R. Radia)

[128]

This is the conference version of the technical report shown in [142], mainly focusing on the UNIX name space adaptation.

When a context (directory) comprises names from different scopes (i.e., the system/lib context of the village and the system/lib context of the machine), the names are merged in order. More local instances are visible first.

File system and NIS name service are not provided by Spring in the UNIX emulation subsystem.

Instead of making all objects be treated as files (the Plan 9 approach), objects that *may* be treated as files support the IO interface.

99 Descriptive Names in X.500 (Gerald W. Neufeld)

[129]

X500 [91] names objects contained in a directory, using their path location (and assumes a hierarchical organization): the name of an object is the path to it from the root of the directory tree. This work tries to assign a better, more descriptive name to objects that's independent of their path names.

A new piece of information, that was not present in [91] is that "first-level" directory service agents, i.e., DSAs that handle subtrees of the DIT right underneath the root, can also handle the root itself. In other words, the 0-th level of the tree is handled by all 1-st level DSAs.

The problem lies in that the organization of the directory (who owns what pieces of it), affects how objects are named. If the directory is reorganized, all names have to change.

Descriptive names fulfill the following two criteria:

- Order of name components does not matter (in comparison, X.500 uses a hierarchical path as a name, therefore name components must be ordered as the tree levels are ordered).

- Not all components must be given (in comparison, in X.500 a missing components means that the path from the root is broken, resulting in ambiguity).

Any number of attribute-value pairs that can unambiguously characterize an object (not requiring that attribute-value pairs belong to the named object only) can constitute a descriptive name. For example, if we want to name person X, and we know noone shares X's office phone number Y, then [office-phone-number = Y] is a descriptive name for office X.

Resolving such a descriptive name can be very expensive. You'd have to search a lot of the entire tree just to locate one object that matches all of the attribute-value pairs in the name. You'd also have to keep looking, just in case there are more matches in the rest of the tree.

Stopping at the first match is not a good solution, since the results of a query would be dependent on where we start from. For example, if I start searching at my local domain, perhaps I'll find a local match before I find remote matches. Different directory user agents would find their local match first. This is unacceptable, since the resolution of a name in the directory should have a single semantics.

The solution preferred is one that restricts the tree space to be searched for a descriptive name. This is done by means of *registered names*. A registered name consists of attribute-value pairs that are protected from duplication outside the owner object. More specifically, if an object X has a registered name Y whose component is [Z=A], then no object within the subtree rooted at X's parent may contain Z=A as one of its attribute-value pairs.

Registered names are used to limit searches for descriptive names within a subtree much smaller than the entire directory. It is assumed that most directory system agents that own (i.e., manage) part of the directory will have at least registered the name of their local root (i.e., the root of the subtree they own). Therefore, if we find the longest registered name within a descriptive name, chances are that registered name lies at the root or within a local subtree of a directory system agent. The search can be more elaborate within that limited subtree, for the remaining name components, which are not registered.

99.0.13 Commentary

Is it clear that subsequent reorganizations of the directory might not render currently unambiguous descriptive names ambiguous?

The designation of an attribute as a "naming attribute" is rather arbitrary. As shown in more recent work (for example, [1]), any kind of qualifier might be useful in a name.

It is claimed that descriptive name resolution is no less efficient than destinguished name resolution. However, it seems to me that the constants might be very different in the asymptotic similarity. Distinguished name resolution involves one traversal of the path from top to bottom. Descriptive name resolution requires one traversal of the registered path from top to bottom, and then an exhaustive search of the found subtree (unless very restrictive heuristics are used to guide the local search).

100 What's in a Name? (Peter G. Neumann)

[130]

This is a single-page article on naming systems (good or bad ones) and why it's very important for them to be well designed.

The most important distinction made is between *identification* and *authentication*. The common practice (mainly off-line) of using an identifier as an authenticator is extremely dangerous and has been proved harmful. Names, even private names (such as Social Security Numbers in the USA), are not as good as passwords: names are customarily used as references (pointers); if one of the places where such a private name is used as a reference is lax about keeping the reference secret, this entire fragile authentication scheme falls apart.

101 Names should mean What, not Where (James W. O'Toole and David K. Gifford

[131]

A short, fairly forward-looking, thesis paper on using content-based naming. The examples are not exactly obvious. Better examples exist now of why

content-based (or intention-based[1]) naming is rapidly becoming preferable. Mobility is definitely one very good example domain.

It's interesting to read pre-web literature. And this is merely 7 years old.

102 Understanding CORBA: The Common Object Request Broker Architecture (Randy Otte, Paul Patrick and Mark Roy)

[133]

(1)CORBA was made to handle multiple architectures, multiple operating systems, multiple data formats.

Connecting components together will happen again and again. Having a way to do that without reinventing the wheel every time would be cool.

(1.4.2)Not only does CORBA handle the data translation between architectures etc., it also locates providers of the requested services.

103 The Profile Naming Service (Larry L. Peterson)

[139]

Another descriptive naming service, like that in [129] or [1]. It's a confederate approach, in that clients are solely responsible for contacting as many name servers as they need; the name servers themselves don't talk to each other.

An interesting characteristic about Profile is that names are allowed to contain limited regular expressions instead of merely literal strings. So, names like "John—[J[ohn]] Doe" can be attributed to an object, to mean any of "John Doe", "J Doe", "John", "Doe".

103.0.14 Namespace Organization

Profile does not enforce a hierarchy unlike most other naming systems. The sole instrument of low-level organization are attribute-value pairs. The named

objects (*principals* in the terminology of the paper), have no relation to each other except for that indicated by an arbitrary attribute.

The use of multiple name servers in the "confederacy" is effected at the user agent. A resolution operation contains the name to be resolved, the search path and a bunch of operational flags.

The name is just a set of attributes. It can contain additional alternation or optionality (is this really a word?) operators. It may also contain the interpret function that should be used to match the given name.

The search path is an ordered list of name server addresses known to the user a priori. This can also be replaced using the indirection operator "@" by a list of names naming name servers (lovely sentence...). In other words, instead of saying "use the name server at address X", you can say "use the name server by name of 'blah blah blah' ". The 'blah blah blah' part is also a Profile name, and the named principal must have an attribute pointing to the network address of a name server.

The operational flags determine whether the first result or all results should be returned, and whether the search should proceed outside the local name server automatically or not.

103.0.15 The Name Server

Name servers can maintain and return referrals to other name servers. Other than that, they have no organization amongst themselves. Any kind of name server hopping is performed solely by the user agent.

The modification API is pretty coarse: users change a public text file and the database is reconstructed (compiled) from it at regular intervals. There's also a protected partition for the portions of the database that should be administered in a more restricted manner. The protected partition contains most complete attributes (e.g., everybody's user name etc.) whereas the unprotected partition contains most incomplete attributes (i.e., things that people want to register for themselves).

There is no defined protection scheme other than that provided by the systems on which a nameserver relies, e.g., the filesystem.

103.0.16 Attribute Soup

Using an attribute-based naming scheme has the following pitfalls associated with it:

- One attribute may describe many principals (e.g., "name = Petros")
- Some attributes in the database may not correctly describe the principals for which they are registered (e.g., "title = Dr." for a non-doctor)
- Not all attributes that describe a given principal are registered for that principal (e.g., a principal who has green eyes might not have registered the attribute "eye_color = green")
- Something that sounds like the second pitfall above. I'm not sure what their difference is in the paper.

Attributes are split into two categories: *complete* and *incomplete*. A complete attribute is defined for all principals in a namespace. For example, if all principals have a "name" attribute, then "name" is a complete attribute. This is similar to the closed world assumption in that, if we know anything at all about a property then we know all there is to know about that property.

Incomplete attributes are considered less authoritative as name components than complete ones. An example of how an incomplete attribute can be a problem is this: Consider principals P and Q and attributes a and b. If a is registered for both P and Q, but b is only registered for P, then a is complete and b is incomplete. Now, if a name contains both a and b, then it is intuitively wrong to not return Q just because b is not registered for it. To the extent of what is know about Q, $\{a, b\}$ describes it just as well as it does P.

103.0.17 Name Matching

In the Profile world, matching attribute-based names can be done either in an *exact* or a *partial* fashion:

A name (attribute set) A matches another B exactly if A is fully contained within B.

A name A matches another B partially if the intersection of A and B is not empty.

103.0.18 Interpret Functions

Using the concepts of complete/incomplete attributes and exact/partial name matching from above, Profile defines a set of interpret functions, aimed at

fulfilling different levels of accuracy or selectiveness. As mentioned above, for example, incomplete attributes are considered less useful to name a principal than complete attributes.

1. The first function returns all principals with some attributes in common with the given name, whether complete or incomplete. This is the least restrictive interpret function.

2. Matches are ordered as follows: complete attributes are used before incomplete ones are used and then exact matches are used before partial matches are used. So, if the complete portion of the name matches exactly the complete attributes of some principals, only those principals are returned. If no such principals exist, then those whose complete attributes are matched partially by the complete attributes in the name are returned. If none such exists, incomplete attributes are used instead, first with an exact match and then with partial match.

3. Briefly, this function uses the same criteria ordering as the previous one. Additionally, it restricts the principals returned by returning those with both incomplete and complete attribute matches.

4. Finally, this function only returns a single principal matches exactly on its complete attributes. If there are more than one matched (in any fashion) they are not returned. This function tries to implement a traditional lookup function.

103.0.19 User Interface

The paper describes a way to bake Profile into a Unix shell by using translation mechanisms that take a standard command (like cp or mail) and yield a Profile-enabled command.

103.0.20 Commentary

This paper is in serious need of a translation. It doesn't look like people who didn't work on it ever read and helped shape its language. The concepts are not always clearly described, the examples are less helpful than you'd expect and things enumerated in the text are never given unambiguous numbers.

104 A Blueprint for Introducing Disruptive Technology into the Internet (Larry Peterson, Tom Anderson, David Culler and Timothy Roscoe)

[138]

This paper describes the necessity for a large, shared testbed for distributed applications that can be used to also deploy tested applications, and proposes the PlanetLab architecture.

The basic necessity behind such a testbed is innovation; the Internet architecture is ossified into one, commonly used thing that people essentially take for granted. It is important, claim the authors, to allow networking innovation at large scales to occur, so as to inform and bias the next Internet architecture. What is needed is a *service-oriented network architecture*.

Such a testbed should be large (on the order of 1000 nodes or more), distributed broadly to capture a larger set of network conditions, allow uniform management of participating nodes, offer a uniform interface to applications it supports, and permit both debugging and testing, as well as actual deployment of services. The authors identify that the two roles of such a testbed—for testing/debugging and for production deployment—are at odds and present unique design and implementation challenges.

Compared to previous initiatives towards testbed design and operation, the authors propose a testbed that is open to all and offers full application support instead of networking only, unlike Internet2; an environment for deployment, not just testing unlike EmuLab; an environment also for small building blocks, not just for massively distributed parallel applications, unlike the Grid; support for applications beyond just forwarding, unlike ABONE, an active networks overlay; and also support for applications and diverse types of networking, not just IP in IP tunneling, unlike the XBONE.

This is an ambitious plan, but it can be more effective that just developing an application and hoping to see it distributed via viral dissemination, because it allows development of "unpopular" applications, it allows iterative refinement of a design even after it becomes popular, and it keeps the environment relatively controlled rather than relying on otherwise unknown hosts to offer their services in a productive way.

The design principles for PlanetLab, the testbed under development,

are slice-ability, distributed resource control, unbundled management, and application-centric interfaces. Slice-ability means that a single application can get a "horizontal slice" of the entire testbed, that is, a small proportion of many or all of the participating nodes in the testbed; this requires some variety of virtualization of participating nodes, and plans for security, resource control and scalability. Distributed control is important in attracting node owners to offer their nodes for participation; however, testbed users may require a global specification of where their applications run. To control these two aspects of control, the testbed must handle the problem of stable credentials, but admission control at the local level. Unbundled management is important in keeping the testbed in a fluid state that can evolve and improve, rather than being ossified in a state that suits most current users. Finally, application-centric interfaces are important because the testbed can only be successful if many diverse and potentially successful applications are developed and even deployed on it; the old adage still applies: keep the APIs stable but change the implementations often.

105 Persistence in the Spring system (Sanjay Radia and Peter Madany and Michael L. Powell)

[141]

This is the conference paper on the persistence-related issues in [142]. The issues in persistence and naming can be reduced to the following:

- What types of objects can be made persistent?

  Persistent objects or persistent classes? Can persistent classes be created? Can persistent objects be shared?

- How are object naming and storage services organized?

  Are both persistent and transient name spaces supported? Are naming and storage independent or conjoint?

- What kinds of inter-object references are supported?

  Are inter-object reference types restricted? Can transient objects be referenced in another object?

- How difficult is storage reclamation?

  Which algorithm (Mark & Sweep, Reference Count)? Is storage reclamation restricted in a single host?

The four approaches handled in the paper include:

The Workspace Approach The workspace approach (as seen in Smalltalk) allows programs to have a persistent environment, and is implemented as a programming environment. Objects and additional persistent object types (classes) can be stored. However, references within the workspace of a program cannot be externalized, therefore sharing is impossible.

The name space is flat and, in most cases, not easily manipulated. The entire workspace is reflected into a single file in the filesystem. Therefore, garbage collection is relatively easy (not to mention that the entire workspace can be collected, by unlinking the file it corresponds with).

Workspaces allow easy manipulation and all kinds of references to be persistent, at the expense of sharing flexibility and programming language selection.

The UNIX approach In UNIX, the only persistent objects are files, directories and symbolic links. Other objects have to be transformed/stored inside a file to become persistent. Sharing among processes/users is permitted, with some limited access restrictions.

Transient name spaces are not supported in the basic UNIX system. Naming and storage are conjoint in the file system.

The only objects that can contain references are directories. No transient objects can be referenced in other objects.

Storage reclamation is handled through reference counting. The method however can change to fit the object types stored.

UNIX supports sharing, as opposed to Workspaces. However, naming is very limited and not at all uniform.

Extended UNIX (Choices) Many object types can be made persistent, in addition to files, directories and symbolic links. Sharing is handled similarly to UNIX.

Transient name spaces are supported, but are handled through a different interface. Naming and storage are combined.

Inter-object references are much more flexible than in UNIX. Still, however, transient objects cannot be referred by persistent objects.

Storage reclamation uses reference counting; algorithms that can handled cyclic structures need to be utilized. Inter-machine reclamation is straightforward, since only symbolic links are allowed across machine boundaries.

General Approach This approach combines the good features of all the above. Persistent objects or classes are extensible and sharable.

Both persistent and transient name spaces are supported. The servers for storage and naming are independent and separate.

References to other objects are unrestricted, and transient objects can be referenced.

Storage reclamation is dependent on stored object types.

Transcription is the service used to transform an object's representation on the client side or an object's data structure on the implementation side into a form that can be used to reconstitute an object in a different location or at a different time.

Time transcription can be tricky, since time related portions of the object (for instance, references to transient objects) cannot be stored. The following are related concepts:

Pickling It's transcription that's suitable for persistent storage. It happens at the server implementing an object and is not directly visible to a client through the object's interface.

Tough problems: following pointers in data structures, handling pointer cycles, handling embedded objects, dealing with inheritance and unpickling a pickled form as a base class of the original object.

Freezing Freezing occurs at the client side, and involves an object representation. Freezing generates a freeze token that the client can use later to reconstruct the object. The reconstructed object is not the same as before being frozen, it shares however a large part of its state. The implementation decides what will constitute a freeze token. A server

may well decide to return a pickled form of the object as a freeze token, or an identifier that will help it reconstruct the object's state later (for instance, the i-node of the file where the object's state has been stored).

Freezing invariably results in the object state being pickled.

A freeze token should be usable by clients other than the one that initiated the freezing. Therefore, no client-specific data should be part of the freeze token (for instance, no door identifiers).

Freezing and melting are performed by the freeze service of the manager of an object. Each object has an Object Manager ID (OMID). This can be stored along with a freeze token. The client that decides to melt this frozen object can locate the freeze service by asking the name server for an object manager with the stored OMID, and accessing its freeze service with the stored freeze token.

Freeze tokens are forgeable. Therefore, the freeze service needs to be authenticated.

Lost freeze token means lost object.

The name service is a good place to perform all freezing operations, insulating clients from them. When no open references exist to an object bound to a name in a name space, then the responsible name server freezes the object reference, storing a name-to-freeze-token binding in stable storage. When a client requests a resolution of a name bound to a frozen object, then the name server melts the freeze-token transparently to the client. Note that clients can always freeze objects directly.

This name service interposition between freeze and freezer, also facilitates access controls: the name server checks all privileges required from a client about to melt a freeze-token. Otherwise, complicated security protocols would have to be observed, to transfer melting privileges from client to client.

The melt operation can be made to start the corresponding object manager (so that it can unpickle an object whose reference is being melted). This can deal with object manager crashes, as well as managers that terminate when they manage no active objects.

Different object require different freeze services:

Data-like objects The representation of the object is the freeze token. Such would be a complex number.

Transient objects They can be frozen either by a per-machine transient object freeze service, which stays up as long as the machine stays up, or by a library inside the object server, which stays up as long as the object's manager is up. The former requires some garbage collection (for those transient objects whose manager has died).

Externalizing It is a transformation intended to create a copy of the object at a different time or location, without any shared state with the original object. Tar'ing a file system would be "externalizing" it.

Files become much less important components of an OS, now. Only applications that need to handle their own storage more flexibly need use files (for instance, to store freeze-tokens, or pickled objects).

Open Issues:

Inter-object references What happens if we freeze an object that contains a reference to another object? Do we freeze the other object and stick the freeze token in the pickled object state? Do we just substitute the object with a name?

Garbage collection How do we delete frozen objects?

- Delete frozen objects when there are no more name bindings for them would mean we'd have to search all name spaces, which can take a while.
- Have the clients of frozen objects "ping" them periodically would incur a lot of pings for a name server, for instance. Furthermore, if a name server got disconnected for a little while, and got back on-line, it might find all its frozen objects deleted.
- Have an owner fro each frozen object that gets charged for resource usage. This way, owners take the responsibility for keeping track of their frozen objects and deleting them when they don't need them any more.

106 The Spring Name Service (Radia, Sanjay and Nelson, Michael N. and Powell, Michael L.)

[142]

Name services can connect names to physical addresses of objects, or to other identifiers (names) which are however easier to resolve (for instance, from human-readable names to object IDs).

Issues in heterogeneous name spaces:

Location How does the glue level get built. How, that is, do the heterogeneous spaces fit together under a common hierarchy (or will it be something other than a hierarchy?)

Results A name service is normally a function from a name to a physical identifier or other name. However, different name spaces have different domains for the result of this function (i-nodes in file-system directories, IP addresses in DNS, file descriptors in STREAMS). How do we enable this diversified functionality under the same roof?

One solution is to make everything look like "something". In most cases, this something is a file. So, give all objects a file interface (i.e., reads, writes, ioctls, and so on.). This is cumbersome, and in some cases impossible. Did I mention esthetically scary?

Dependency If the manager doesn't implements the name service, then it knows how to handle persistent or transient object references and names. In those cases however, the name service is not uniform. To make the name service uniform, we must detach it from any one object manager, and give it ways to handle persistence with the object managers whose objects it names. Another way to do that is to resolve names to other data IDs and then submit those IDs to the object managers for further processing (as opposed to having an object representation).

In SPRING, name spaces and contexts (a generalization of a directory) are first class objects; they can be shared and copied and duplicated and destroyed and combined and stuff.

Object references are not persistent (you can't "save" an object "pointer" - you need to reacquire the pointer after execution is resumed).

Access control is performed at the name server. If the object manager wants to implement it itself (for instance, it doesn't trust the name server) it can do that in addition to the name server's controls.

Names contain different fields with status information, version, and type (C program or object code, and so on).

There are three kinds of persistence:

Object state How the object manager stores the current state of an object, when it is, in general, not active (i.e., not referenced by any clients).

Object access In SPRING, there are no persistent object pointers (as in Clouds for instance), since not all objects will be persistent. In Clouds, there are object IDs which remain persistent even when the object moves around. They're pretty much names, without the "readability" of those we customarily consider names.

Name binding When an object has a name binding for it at a name server, then it is persistent. Whether it's active or not is generally determined by whether there are references to it or not. Otherwise, it can be "frozen", i.e. turned into a data value (like a marshalled, flattened version) and stored. This is normally what happens at the name server for the object references of objects handled by object managers.

Access is only checked when trust boundaries are crossed (i.e. when we're moving to a name server for a context that's not collocated with the previously checked name server). If at some point the existing authentication is not enough for the current name server (because we crossed a trust boundary), the original client is asked (through a raised exception) to authenticate itself with the name server directly. Even when a context is passed between domains owned by different principals, the exception will be raised when the current credentials (in the capability within the context) don't match the current principal. In those cases, the client owned by the different principal will be contacted to authenticate itself directly.

Freezing and melting are done outside the object.

"Virtual worlds" can be created, with slightly altered hierarchies (say when a new release of the system is tested and only some of the executables have changed). In those cases, domains can run with the altered root context

and carried name space, without any indication that this is not the world that the rest of the domains run under.

107 Naming Policies in the Spring System (Sanjay Radia)

[143]

This is an elaboration of the naming policies in Spring, as seen in the technical report([142]).

In the case of remote domain invocations, children inherit namespaces that contain all ancestral local machines and villages, under the contexts with names `local_machine_X`, `local_village_X`, where X is an integer increasing down the inheritance path. The home contexts are those for which $X = 0$. The currently local contexts are those for which X is maximum, and they are pointed at by the symbolic links `local_machine` and `local_village`. The parent, however, does have the option to keep the local contexts that the child inherits unchanged; in those cases, the child executes in its parent's naming environment, though remotely.

108 A Model, Analysis, and Protocol Framework for Soft State-based Communication (Suchitra Raman and Steven McCanne)

[144]

A very dense, very impressive paper (received the best student paper award in SigComm '99), modeling and analyzing soft-state protocols.

Soft state was first coined by Clark in a SigComm '88 paper on the design philosophy of Internet Protocols, though not quite fully defined. What makes state soft is that it can expire. As long as there are refresh actions keeping it fresh, the state remains. When the refreshes cease to appear, the state eventually goes away. This makes recovery under failure much easier than having to deal with explicit, "hard" state that requires active restitution in the face of failure.

Soft state requires extra messages, since it relies on refresh.

The Internet is the perfect domain for soft state protocols, since it contains endpoints with a wide range of failure rates, reliability guarantees and performance characteristics. Soft state-based design fits the Internet better, since it calibrates the design process to anticipate conditions that occur quite often on the Internet.

This paper proposes a model for soft state protocols and defines a consistency metric on it. It also suggests a soft state-based transport protocol framework as a building block for the design of soft state protocols.

108.0.21 Data Model

The model assumed is one where the sender maintains a database of attribute-value pairs $k, value(k)$ and periodically announces those on a channel, from which receivers can extract pairs and insert them or update them in their local databases. When a pair expires on the receiver, it gets removed. Additionally, pairs can be removed from the sender, which causes the sender to stop refreshing them.

The consistency metric $c(k,t)$ for an item k at time t between sender S and receiver R is defined as $c(k,t) = Pr.[S.val(k) = R.val(k)], 0 \leq c(k,t) \leq 1$, where $P.val(k)$ is P's value of k. Given this definition, *instantaneous system consistency* can be defined as the average consistency over all items in the database at time t and, similarly, *average system consistency* can be defined as the time average of the instantaneous system consistency over a long time period. A system is *eventually consistent* when the instantaneous system consistency approaches 1 a long time after the last item has been inserted in the sender's database.

Periodic updates are good enough to foster eventual consistency over a lossy medium, but they can be expensive (redundancy) and can cause undue propagation delays.

108.0.22 Model Analysis and Simulation

Unfortunately, most of this section went right over my head. However, the gist of it describes a simple refresh scheduler (FIFO) and analyzes its performance and bandwidth wastage. The simple FIFO scheduler maintains system consistency between 85 and 95% for loss rates in the 1-10% range (with a 15% probability that value-attribute pairs will be removed from the

sender). For loss rates as high as 50%, and pair removal probability of 10%, bandwidth waste is calculated around 90%.

A better way to manage refresh transmissions is to use the tried and true technique of ageing data. Retransmissions (refreshes) are considered less important than new transmissions (of new attribute-value attributes or updated ones). Therefore, there now exist two transmission queues at the scheduler, a "cold" one (refreshes) and a "hot" one (new). Through simulation, the authors have shown that when the bandwidth used by the hot queue is slightly higher than the rate of update at the sender, system consistency is maximized. Furthermore, update latency decreases as the bandwidth used by the cold queue is increased.

To make retransmissions more effective, a receiver feedback mechanism is added in the scheme, where detected losses by the receiver create a negative acknowledgement. When an item is declared missed by the receiver, it is moved into the hot queue again by the sender. In other words, the hot queue contains not only new transmissions but also requested retransmissions; the cold queue contains only spontaneous refreshes. When the bandwidth used by feedback traffic (NACKs) is between 20 and 60% of the total bandwidth, consistency reaches almost perfect levels.

108.0.23 The Soft State Transport Protocol framework (SSTP)

This is just a nice, packaged form of the theory derived in the first part of this paper. It allows applications to specify what levels of consistency they expect from the tranport layer. Then SSTP arranges the bandwidths used by hold and cold queues to achieve that consistency. It also relies on receiver reports to be informed on the observed loss rate. Finally, it requires some separate module to tell it what the available bandwidth is.

SSTP provides upcalls to the application, to allow it to change its behavior when the consistency "guarantees" are not maintained. Such is a case when the application feeds data into SSTP faster than the bandwidth it has allocated for the hot queue.

To deal with large data spaces, SSTP allows applications to put their "soft" data in hierarchical name spaces. Transmission and update of such name spaces is done in a recursive descent-based manner. In this way, some receivers might decide to skip parts of the name tree (and, therefore, not request those parts).

108.0.24 Commentary

Despite the importance of the contained work, this is a really sloppy paper, apparently not very carefully proofread. Especially the analysis sections (3-5) lack basic continuity, like explaining what some of the symbols used signify and the like. The graphs are minimally captioned and not adequately explored in the text. There's a figure that's never even mentioned in the text. This is a real shame, since this is an otherwise extremely cool paper. Pity.

109 Crowds: Anonymity for Web Transactions (Reiter, Michael K. and Rubin, Aviel D.)

[145]

This paper describes how browsing the web can become an anonymous task. This is roughly based on the mix approach[43]: participants in the scheme form *Crowds*. Every participant in the crowd runs a browser and a *jondo*, a mix server. Before a web request from a participant's browser is delivered to its intending server, it is sanitized (all identifying information in it is stripped), and then directed through a number of jondos, before the last one delivers it to the destination web server. Every forwarder has a fixed probability of doing away with forwarding and just sending in the request instead. Responses from the web server a forwarded back using the reverse path through multiple jondos.

In the process of substantiating the validity of their design, the authors describe an interesting semi-continuous anonymity metric. They identify the following points on the anonymity scale:

Absolute privacy An attacker can in no way distinguish the system states in which the attacked participant actually participated in communication and those in which he did not. In other words, any communication where the target of the attack participates produces no observable effects for the attacker.

Beyond suspicion Though there is evidence that a message has been sent, the attacked participant appears no more likely to be involved in the communication than any other potential sender in the system.

Probably innocence Though the attacker may have reason to suspect that the target is involved in a communication, the probability that the target is not involved is still higher.

Possible innocence There is a non-trivial probability that the target is not involved in a communication.

Exposed The attacker can identify without reasonable doubt that the target is involved in a communication.

Provably exposed Not only does the attacker know that the target participated in communication, but he can prove this to others.

These degrees of anonymity apply to senders, receivers or even the link between a sender and a receiver.

Crowds has the following differences from the original mix scheme in [43]:

- They offer probable innocence for senders, instead of the sender/receiver unlinkability of mixes
- They offer no protection against global eavesdroppers
- They do not rely on public-key encryption
- They do not restrict message length
- They do not use any kind of traffic analysis decoys
- Their throughput grows as a function of the number of users, where using mixes throughput is proportional to the throughput of every mix, and decreases linearly with the number of users

Although the jondo must know a membership list of the scheme, this list need not be up-to-date for crowds to work. The list's accuracy does affect performance.

Once a path from a web-browser to its intended destination web-server has been formed, it is used consistently for all other requests or responses traveling between the same endpoints. This path does not change, unless one of the intermediate jondos fails. Keeping the path static helps avoid sender tracking by colluding jondos.

Paths might also change when new jondos join a crowd. To avoid singling out newcomers, joins are batched. Once a batch of joins is committed, all

paths a flushed and recalculated – otherwise, new path formations could be easily attributed to the newcomers.

Corrupt jondos can also use timing attacks to infer the identity of the browser, by timing how long it takes for a new request, following a previous response to arrive to them; if the time is similar to one round trip time between them and their preceding jondo, they can infer that the latter is the initiator. Such tightly spaced HTTP requests would result from embedded images. To avoid this problem, in crowds embedded requests are issued to the web server by the last jondo in a path, not by the browser.

Crowd membership is centrally managed by the *blender*, a trusted third party. The authors do not foresee any problems with making the blender a distributed mechanism.

110 RFC1484. Using the OSI Directory to achieve User Friendly Naming (OSI-DS 24 (v12)) (S. Hardcastle-Kille)

[79]

This scheme is intended to provide user-friendly naming within a larger directory service, such as X.500 [91]. The distinguished name of an entry is separate from the user-friendly name for the same entry. The distinguished name is a unique identifier, whereas the user-friendly name may omit features of the entry that don't interest the requester. The scheme is heavily dependent on X.500, which is in turn heavily dependent on organizational structure. User-friendly names are reduced to their structure using defaulting rules, for instance, when the inferred country is USA, then a two-letter component preceding it is assumed to be a state.

There is a long list of heuristics used to resolve a user-friendly name to a distinguished name, assuming that the user-friendly name follows certain conventions, such as hierarchical ordering. Perhaps in current technology that deals with searches of large distributed data stores, such heuristics wouldn't necessarily be of much use.

110.0.25 Issues

One important advantage of this scheme is that, similarly to X.500, it doesn't have to invent a new name for an entry. Instead it uses the commonly used name for an entry. Unfortunately, some entries may share the same "commonly used name", for example roles of the same person, or people with the same name. To resolve such ambiguities, much longer names would have to be devised.

It seems that this scheme is merely a glorified method for resolving a local alias to the globally unique name for an entry. The resolution takes place locally and might come across ambiguities when transferred to a third party.

111 RFC 1546: Host Anycasting Service (C. Partridge and T. Mendez and W. Milliken)

[134]

This deals with the concept of picking one Internet host among many of the same class, based on some application-specific characteristics.

Should this be implemented at the IP level, there are several important and difficult issues to deal with:

- If we're using an *anycast* address to access a "virtual" host, how do we make sure that all packets in the same flow are delivered to the same member of the virtual host group?
- How are anycasting addresses assigned? Carving out a class of addresses out of the severely crowded IP address bit space would impose additional limits on address distribution, making it easier, though, to route anycasting packets. Ad-hoc designation of addresses as anycasting addresses makes a more efficient use of the IP address space, but makes routing extremely difficult (since there are no clear ways to mark an address as an anycast address).
- State-ful protocols, like TCP, cannot rely on anycast for continued communication with another TCP stack. The best way to deal with this is to replace the anycast address with a unicast address (the address

of the first SYN-ACK received) and proceed to communicate with that host using unicast.

- Anycast packet routing on local networks (where a router cannot be assumed to exist) have to rely either on proxy ARP (i.e., an anycast participant has to respond to an ARP request for the anycast address), or on link-level multicast addresses. The ARP approach will cause more ARP response transmissions (since the caches must prediodically be refreshed), whereas the multicast approach will cause all anycast packets to be received by more than one hosts.

111.0.26 Commentary

This is an early attempt at "what"-not-"where" addressing and naming. Obviously, the criteria selecting members from anycasting groups can only be those understood by routers (hop counts or link bandwidths at the most). For more advanced criteria, application-level smarts will be required. Furthermore, the hackiness of it all is just mindboggling.

Bottomline is, too much overloading (in the OO sense) can be bad for your designing health.

112 Can we eliminate Certificate Revocation Lists? (Ronald L. Rivest)

[147]

The point of this paper is to explore a world without Certificate Revocation Lists.

CRLs are there to deal with the painful fact that certificates become obsolete: either their signing key expires or gets compromised, or the fact they certify is no longer applicable. For example, if a business X, Inc. has issued a certificate that Petros is an employee of X, Inc., there must be a way for that certificate to no longer be considered valid once I quit X, Inc., or X, Inc. fires me. In their original form, CRLs are just issued every once in a while by the Certification Authority.

The paper justifies a series of propositions about certificates and their expected usage patterns:

- It is the certificate acceptor who knows best how "fresh" a certificate should be, not the CA.

  This proposition supports that any arbitrary refresh period picked by a CA for new CRL issuance is not going to fit the freshness needs of all possible certificate users. The only way to provide as much freshness as needed is to have an extremely high CRL issuance frequency, which can be very demanding on network resources for CRL dissemination.

- It is not the certificate acceptor's business to dig out all relevant certificates for a certification task; that is the business of the certificate provider, the client of the requested service.

  The author justifies this proposition by the fact that certificate acceptors are probably busier than certificate providers; and that acceptors can in that way be implemented in a more stateless fashion, since if they deem a bundle unacceptable, they can just say what is wrong with it and forget about it.

- The best "recency evidence" for a certificate is a more-recently issued certificate.

  This makes sense, especially when the certificate is not excessively large, which should be the norm. The only catch is that the certificate issuer, or a duly authorized representative, must be online at all times, to supply those fresh certificates.

- Certificates should have a time period when they are absolutely guaranteed.

  The author extends the usual certificate guarantee statements into a more general certificate guarantee. According to the latter, a certificate has a period when it is absolutely, positively valid, no matter what happens in the rest of the world; the issuer has the knowledge and the right to specify that period. Then, the certificate has a period when it is probably valid, but revocation is possible; during that period, paranoid acceptors should seek a reconfirmation. Finally, the certificate becomes invalid outright and should not be accepted under any circumstances.

- Certificates' validity is separate from the validity of the signer's key

Instead of invalidating a certificate once the key of its signer has been invalidated (through compromise or expiration), according to the author, acceptors should require separate proof that the signer's key is still valid. This proof can come from public, trusted *suicide bureaus*, which keep track of invalidated public keys and give out certificates of health. Of course, this requires global trust on a network of such bureaus, which is not considered acceptable by everyone.

This last issue sounds a bit odd. It isn't abundantly clear what exactly the semantics of a valid certificate issued by an invalid key is. Besides, isn't this just going to open up the implementation nightmare of ensuring that all acceptors look for a bill of health for every signer whose certificates they validate? Admittedly, this could end up being a reasonable tradeoff, in cases where multiple certificates signed by the same principal are used in the same computation.

113 The Design and Implementation of a Log-Structured File System (Mendel Rosenblum and John K. Ousterhout)

[148]

Main assumption: read requests are easily satisfiable by using main memory; optimize for writes and many small files. As memory becomes more abundant and less expensive, more of it can be available as a file cache (for both reads and writes).

Gripes against older FSes:

- too many seeks
- too many synchronous writes(metadata)

To fix this, writes are performed sequentially in a log. Once enough write data are available, a single write operation into the log is initiated, requiring only one seek. This includes data and meta-data blocks.

Major issues:

- how do we retrieve data?

- how do we find consecutive empty blocks?

The log is the main storage structure, i.e., it's not temporary. So, every time a file changes, it and its i-node get written in the log. An in-memory i-node map points to the current location of all i-nodes. This map gets checkpointed on disk periodically.

For writes in the log to be fast enough, contiguous long writes must be possible. The disk is split into 512k segments. Segments are always completely overwritten, if changed at all. Segments with little live data are copied over in a compacted form.

Liveness is determined by checking to see if a datablock is pointed to by the appropriate pointer in the associated i-node (the i-node number and the block number for each block in the segment is stored in the segment's summary).

The *cleaner* is the component that compacts live data to open up more contiguous space (more empty segments) for the log. It attempts to only read in segments with a low liveness ratio, to minimize the cost of cleaning. In addition to the above criterion, recently modified blocks are considered likely to change soon, i.e., not worth compacting. So, the age of the live blocks in a segment is also taken under consideration when deciding whether to clean or not to clean.

Crash recovery uses checkpoints and roll-forward. A checkpoint contains the i-node map, the segment map and a time stamp. Given this initial info, segment summary blocks are used to roll-forward the FS with operations that happened after the checkpoint.

LFS is slower than a traditional fs only when reading sequentially a file that was written randomly.

Main performance factors: cleaning(about 25%), rw(75%). In UNIX it's seeking(90%) and rw(70%).

114 Evolving the Vnode Interface (David S. H. Rosenthal)

[149]

The general idea is pushing the vnode concept [98] further, by endowing with inheritance its object oriented model.

16 trillion levels of indirection, since v-nodes can now stack, fan out, tree... All operations have to be called through a whole chain of invocations (one to go to the Vnode, then to go to the top of the stack, then to follow the head of the stack one level down and then to indirect through the operations vector).

The description sounds cool structurally, but I'd have to see more convincing performance results than those included in the paper before making judgment.

115 Trusted Distribution of Software Over the Internet (Aviel D. Rubin)

[151]

This paper addresses Betsi, a tool for the secure distribution of software over the Internet. The approach is based on a transitive trust model through the use of a central trusted identity service. A software author authenticates himself with the trusted identity service, which keeps track of associations between public keys and real-world identities. Then, to publish software, the author signs the software with his private key and forwards it to the identity service. The service verifies that the signature on the software is correct and then signs the software itself as having come from the author.

The paper describes the notion of trust in the software as the combination of the user's trust of the author as a good software producer, and the user's trust that the software came from the author.

116 Design and Implementation of the Sun Network Filesystem (Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, Bob Lyon)

[152]

This is about one of the most popular remote filesystem on the Unix platform.

Design goals:

- no dependence on OS (allow for heterogeneous client accesses)
- crash recovery
- access transparency
- UNIX fs semantics

Built on top of SunRPC. Calls are idempotent, so lack of UDP reliability is no problem; retries can be used safely. "Stateless" means that all necessary information for a call is given inside the parameter list. The protocol is stateless, not its components (i.e., the server keeps state for the clients and each client keeps state for the server).

The fhandle(file handle) represents the server's FS-specific identifier for a file at a client, and it's opaque as far as the client is concerned.

The file access protocol (NFS) is different from the filesystem mount protocol. Mount protocols need OS-specific information to put a client-server association together, whereas the file access protocol is OS-unspecific.

Statelessness implies synchronous disk operations on the server. In other words, all modifications on the server are flushed before a call returns.

i-nodes are versioned, to avoid i-node aliasing (i.e., using a stale file handle on an i-node that has been recycled).

NFS motivated the design of v-nodes, as a way to provide uniform access to different filesystems inside the kernel.

Both protocols (NFS and MOUNT) were implemented in user space. The kernel vnode implementation of NFS jumped out of the kernel to have requests completed.

The hard issues on which the paper comments are, strangely enough, those which still haven't been fixed (give or take one or two).

- The whole issue of access control and root squelching. The same hacky solution they found (making root less powerful than normal users) is still in place.
- Concurrent access. Still a mess...
- Locking.
- Open file semantics. Since the server can't maintain state, it can't just open a file, unlink it and keep on using the file pointer. They fixed this with a hack.

- Modification times and time skew. Still a problem, although not exclusive to NFS.
- Performance: Horrendous, since everything is going through RPC, XDR and the whole shebang of intermediate layers of stuff.

All in all, a great idea but we should have moved along quite a bit since.

117 A Hardware Architecture for Implementing Protection Rings (Schroeder, Michael D. and Saltzer, Jerome H.)

[154]

Hardware considerations for a ring protection mechanism.

Criteria to judge access control mechanisms:

Functional capability (or functionality) A mechanism must to "meet interesting protection needs", which can be accomplished by the primitives provided, and do so "in a natural way", which qualifies the user interface.

Economy (or performance) The cost of specification and enforcement should be low enough not to be a concern in the selection of a mechanism. The actual cost should be proportional to the functionality actually used, not to the functionality available.

Simplicity Lack of simplicity often means lack of security (since complex systems are tough to understand and test).

Generality The internal structure of components should be independent of external access controls.

The above criteria are usually combined by keeping the latter three constrained within reasonable ranges, and trying to provide as much of the former.

In this work, the classic Multics segmentation architecture is assumed, ignoring paging.

Protection rings were motivated by the need to change a process' privileges, as it is executing different code. Note that in Multics, a process is

loosely equivalent to a user shell. Different segments are equivalent to programs/processes in Unix.

A protection ring is a generalization of protection domains. Instead of only having user and supervisor domains, there exist now r domains totally ordered in terms of the privileges they convey (0 being the most powerful). Each process executes in a single ring at any time. Each segment defines a privilege bracket for every possible right (*read*, *write*, *execute*) when the right is enabled by the associated flag. A process has the right if and only if its ring lies within the associated bracket for the segment accessed.

A process can increase its ring freely, since ring increase means monotonic reduction of privileges. Ring decrease, however, is heavily controlled, since it imparts additional privileges. Special locations within a segment are defined, called *gates*. A process can raise its ring number to that necessary to execute a particular segment only when it jumps directly into one of those gates. The ring decrease is only allowed when jumping into the gates from a particular range of rings (called the *gate extension*). The gate extension effectively describes the rings from which a downward jump into a privileged segment is allowed. For rings above the gate extension, a downward jump is disallowed. Gates and gate extensions are stored in the descriptor of a segment and are originally read from the access control list associated with a segment on storage.

Execute and write rights always overlap only in a single ring, if they are both allowed on a segment. This makes sure that only the ring with the highest execute privileges is allowed to write a segment.

For an example, consider a segment A with a $0-3$ write bracket, a $3-4$ execute bracket and a $+2$ gate extension. A process in ring 5 may jump into one of A's gates, and lower its ring to 4, since ring 5 is within 2 of ring 4 (the top of the execute bracket of A). However, a process in ring 7 may not jump into one of A's gates (and therefore, may not lower its ring through A), because ring 7 is not within 2 or ring 4 (the top of the execute bracket of A).

Calls and returns can cause problems when the protection domain changes between caller and callee. Caller-callee environments must be independent (i.e., the environment of lower-ring segments should not be accessible by higher-ring segments). For an example, consider caller-callee stacks should be independent: Stack records written at ring n should not be accessible at ring $m > n$. Otherwise, consider segment A executing at n and calling segment B executing at m. If B has access to A's stack records, it can alter them, thereby causing B to return to code it didn't intend to execute. This

is solved by having different stack segments for each ring, making sure that the stack segment for ring n is not accessible at any ring $m > n$.

In the case where the call lowers the ring number, the callee must ensure it doesn't rely on the caller for its correct operation (since the caller is, in fact, less trusted). The caller must make sure it doesn't allow the callee to circumvent trust boundaries:

- The callee must be able to locate a stack area without relying on any information received by the caller. This is accomplished by reserving specific segment numbers for related ring stack segments (e.g., ring n has its stack in segment $f(n)$). A special word in each segment (reserved by convention) stores the current stack pointer offset for that ring.

- The callee shouldn't allow the caller to use as arguments for a call data the caller couldn't access in the first place (for example, a caller at m shouldn't trick the callee to use arguments accessible at $l > m$, even if the callee executes at $n > l > m$.). Therefore, the callee must validate arguments against the caller's ring number, to make sure they are legitimate arguments given the caller. The hardware allows the association of a higher ring number with specific instructions. Therefore, if the callee knows the caller's ring number, it can access the arguements using this more restricted ring number.

- For reasons described in the previous problem, the callee must know the caller's ring. This is also important when returning from the call, since the ring should be restored to no lower than what the caller had. This is solved by causing the processor to put the ring from which is switched in a process-accessible register.

Calls at the same protection domain work exactly as described above. However, upward calls also have problems.

- It's possible that the caller's arguments are inaccessible by the callee. Some possible HW solutions are described but are all unpleasant and clumsy.

- The return jump must proceed through a gate (since it advances from a higher to a lower ring). Such gates should be only used for the return jump (i.e., not for other jumps which could be performed further down the calling stack). This means that gates should be managed in a stack-like fashion. This is beyond what HW could do.

To fix the problems above (which seem to be difficult to fix correctly in hardware), upward calls trap into the OS, which sets the environment accordingly.

A suggested ring organization for Multics would be:

- Hard-core supervisor routines (access control, I/O, virtual memory, scheduling) on ring 0.
- Less critical supervisor routines (accouting, ...) on ring 1.
- Procedures with access to supervisor routines (through defined gates) on rings 2-5. At this level, protected user-level subsystems could live on rings 2 and 3. All normal user processes would run on 4. Therefore, common shared libraries on 2 and 3 could be protected, without the need to be included in the supervisor. Ring 5 could be used for debugging (i.e., self protection).
- Procedures without access to supervisor routines on rings 6 and 7. Such would be controlled-environment procedures, being checked by user-level procedures running on 4. For example, a generating procedure on 6 could be tested by an evaluator running on 4.

Protection rings don't deal with "mutually suspicious programs", running with the same amount of privileges over a trusted arbiter.

117.1 Experience with Grapevine: The Growth of a Distributed System (Michael D. Schroeder, Andrew D. Birrell and Roger M. Needham)

[153] Experiences and lessons learned on the system described in [27].

An interesting point is that Grapevine had an upper limit on how scalable it should be: 30 servers and 10000 users tops. It is useful to know these numbers before the design phase.

A significant scaling problem faced the system due to global distribution lists, whose membership roughly remained proportional to the user population. Since groups are expanded fully at a message server, before a group message is dispatched, message addressed to global groups could result in excessive latency. One way to deal with that would be to add greater indirection (i.e., intermediate lists within the list), and avoid total expansion of group memberships. For example, a global list "news.global" could be further

partitioned into "news.west", "news.east", etc. In this way, message delivery to "news.global" would be offloaded to message servers for the "west" and "east" (and so on) registried, parallelizing the task.

Another interesting point has to do with the concept of a moderator in distribution lists. Regardless of what the system can or cannot do fast enough, people start getting overwhelmed when, at a point, they start receiving large numbers of messages.

Duplication of servers (message servers in this case) is not only a solution for overloading, but also for availability. It's good to have a local server, dealing with unreliable connections, instead of making the lack of reliability visible to end users. Registry replica location and distribution is dependent on five factors, according to the authors:

- Put the data where it's going to be needed: close to the message servers containing inboxes for the registry
- As before, registries are needed to expand distribution lists: put the registry server for a group close to a message server accepting messages for that group.
- Put registries on either side of unreliable links.
- Have enough registries to ensure no data loss (from unexpected disk crashes).
- Avoid overloading.

A detailed description of the administrative facilities of the system is provided in the paper.

117.1.1 Backlash of Transparency

There are several problems arising from transparent replication. The paper (in section 6) puts together a list of bad design decisions that lead to problems, with regards to replication transparency. Below I'm abstracting the design decisions into a list of things not to do in distributed systems design.

The Synchronous Operation Metaphor Operations that take a while to settle (such as the registry update operation in Grapevine) have an unknown completion time. However, clients (may be lead to) believe

that an update is completed when the first server returns from the update operation. Either a completion event must be generated when the update is done, or operation semantics must be modified to convey the rather user-unfriendly task that they're really accomplishing.

Distributed Statistics Distributed systems don't have to be unobservable. Statistics about the workings of a complex system are very useful for that system's maintenance and debugging.

Growing Pains Unoptimized operations which don't happen frequently are sometimes ignored in unitary systems according to the rule "optimize for the common case". In distributed systems however, if previously slow operations can become simply unmanageable (because of network latency, scale changes, fussy failure modes). It's important to figure out exactly how infrequent an operation is before discarding it as something that doesn't deserve optimization. A good rule: *Sequential lookups are never good in distributed systems.*

Global Knowledge Assumption Assuming that knowledge on the entire system will be momentarily available (locally or within reach) is highly miguided. Algorithms should be designed taking under consideration transient failures and planning ahead for half-way solutions that make do with what's available.

Operation Equivalence Assumption Assuming, right off the bat, that reordering the components of a distributed operation (which may be cheap in the unitary world) leads to equivalent operations is grossly optimistic. Similarly, arbitrary choice of the location of different modules in a distributed operation can also lead to inefficiencies. In the distributed world, the cost of shipping things around becomes quite more significant than simple local bus operations. For example, consider task "A then B", which consists of subtask A, which takes a lot of input and produces little output, and subtask B, which takes little input and produces a lot of output. If I run A far away and B nearby, I have to ship a lot of input to A's site. If I run A nearby and B far away, then I only have to ship little data to B's site, after A is done. Obviously, the latter setup is preferrable.

117.1.2 Modes of Operation

There are several problems arising from unexpected uses of Grapevine. The paper (in section 7) puts together a list of encountered problems. Below I'm abstracting the lessons learned

Undue Generality The designers of Grapevine attempted to simplify their APIs by, for instance, providing only one way to perform updates: send the new value and let the recipient merge it with its old value. However, there are cases, especially in distributed directory services, when such generality is not helpful: incremental updates should be allowed, to facilitate updates of BIG values. Simplicity is good, oversimplification is not (and that's a horrible simplification of the issue, of course).

Unexpected Operation "When you have a hammer, everything looks like a nail". Using a system for something it wasn't intended can lead to suboptimal performance. For example, using a naming system essentially meant for personal, human-timed communications to provide naming in remote filesystem access can be unhealthy. Operations may take long enough to frustrate a fileserver, but not long enough to frustrate a human user. Similarly, if a system is used for random accesses, when designed for sequential access, neither the system nor the users will be happy.

117.1.3 Reliability and Robustness

Here are the lessons learned from the authors' experience in the areas of reliability and robustness:

Load Planning It is offen difficult to plan against overloading contingencies. Commonly, when unexpected pathological overloads occur, cascaded failures due to overloading might ensue. Planning against domino effects is a good thing.

Service Quality Guarantees To allow for service differentiation, service discovery algorithms must be selective: not all services are for everyone. In the example here, if a server is intended as a low-latency, high-availability server, it shouldn't serve inadvertently clients not authorized for it. The maildrop facility in Grapevine is a loophole in service differentiation, causing any kind of selective resource discovery to fail sooner or later.

117.1.4 Commentary

The solution to the global distribution list problem is inadequate for a more scalable system (i.e., one that doesn't such a low upper scaling limit as Grapevine), since it relies on administrators figuring out what a good partitioning is, by themselves. Such configuration tasks are better done automatically, especially if a system is to grow and scale gracefully.

(In section 7) Using expired cached credentials in the absense of connectivity is extremely disconcerting. How do we break a file server's security? We yank the network cable out and then we bruteforce its capabilities.

It is a common implied stance in this paper to assume that possible bad things that haven't happened so far won't happen in the future. Invariably, the bad things in question are pathological conditions caused by excessive workloads. It is quite naive to assume such conditions won't occur, should the system is fully deployed, just because they haven't in a limited deployment of the system. At least, this stance is never spelled out in the paper.

118 Network Event Recording Device: An Automated System for Network Anomaly Detection and Notification (David G. Simmons and Ronald Wilkins)

[156]

NERD is a suite of programs that allow the collection and monitoring of network events over a LAN or larger installation. NERD utilizes the UNIX syslogd facility to communicate events between hosts in a network, which is independent of specific platform architectures. Authenticity of event streams is based on IP address access control lists.

119 Security Considerations for Peer-to-Peer Distributed Hash Tables (Emil Sit and Robert Morris)

[157]

The authors consider distributed hash tables and identify possible security concerns at the routing and storage levels, as well as other "miscellaneous" concerns. The authors further constrain the attacks they consider by excluding those that cause the system to return "incorrect" data; they claim that self-certifying data solve that problem. Instead, the authors focus on liveness concerns, emphasizing the value of *verifiability* of any information received from unknown system participants.

Routing To avoid message forwarding malice, the authors suggest checked iterative routing, which requires iterative routing in the first place. The sender must have a reasonably accurate checker for identifier space proximity to the destination of a message. When the returned next hop from a peer appear to be farther away from the destination than the forwarding peer, then the sender should suspect foul play and try alternate routes. Similarly, the next hop suggested should recognize that is should be the next hop (assuming it is not participating in the malicious conspiracy). A centrally controlled node identifier space seems to be necessary for such solutions; if enforced, it can also shoulder the burden of a public key infrastructure, which would simplify many of the necessities of secure routing.

To avoid malicious or incorrect routing updates, the authors recommend the use of heuristics or format checkers to constrain peers' routing tables (assuming that node identifiers are also controlled). This is harder to accomplish with optional routing updates, such as those needed for locality optimizations due to round trip times.

Attacks akin to routing update attacks seek to partition an existing system, or to pull unsuspecting peers into a separate, adversary-controlled network. Such attacks can be mounted by malicious bootstrappers (who have great power for mischief in any peer-to-peer system), or by peers on the bootstrapping path of a new node. Even the very task of identifying trustworthy bootstrappers can be tricky when node identities are easy to control of forge.

Storage A distributed hash table is open to storage-oriented mischief generally categorizable as repudiation of responsibility; for example, peers can accept data for storage but refuse to serve those data to clients. Addressing such attacks seems to require avoiding single (or "narrow") points of responsibility, for storage, replication, replication enforcement, etc. Interestingly, properties of systems intended for improved performance are at

odds with security. For example, placing replicas or neighboring nodes can yield uniform access but reduced resilience to faults. Instead, replicas should be placed on independent nodes. Unfortunately, disparate node identifiers do not necessarily indicate node independence, since node identifiers, especially within relatively small populations, can be biased productively for attackers.

Miscellaneous Among many "miscellaneous" security concerns, the authors suggest inconsistent behavior (i.e., presenting a good face to part of the network and a malicious one to another part), overload for denial of service, high churn (i.e., rapid joins and departures from the overlay), and unsolicited messages ("spam") as important ones.

Inconsistent behavior is hard to counter in the absence of unrepudiable message authenticity machinery such as digital signatures. Even with signing, slander is a problem.

Overload is, in general, not a great concern for peer-to-peer systems. The spread of nodes over a generally uniform name space helps a lot with load balancing. Again, this uniformity assumption can result in problems if broken.

The authors acknowledge the seriousness of the churn problem. Even in non-malicious situations, rapid joins and departures from the overlay can reduce the efficiency and performance of the system, especially when it involves data migration (e.g., to new replicas replacing departed peers).

Finally, unsolicited messages can be used for multiple goals, including denial of service. The best defense against this concern is to use challenge/response unicast messages to ensure that the intended recipient is participating in the exchange (assuming that eavesdropping is not possible).

120 Input/Output Optimization and Disk Architectures: A Survey (Alan Jay Smith)

[158]

A very interesting survey, supported by a large number of large enterprise installations. The issues brought up are somewhat outdated, but a large portion thereof still have a great significance.

Why is I/O an issue? Why do we want it optimized?

- Gap between main memory and fast disk access times is enormous (about 5-6 orders of magnitude).
- CPU has to do a lot of stuff to issue and get a response to an I/O request

Block Size Small block size on disk means:

+ smaller buffers are required
+ buffers are quicker to transfer, once they're located
- data are scattered into more blocks, which have a greater probability of being in more different places on disk
- the physical inter-block gaps on disk waste space

Data Set Placement

- Related data should be placed close together
- Frequently used data should be near the middle of the disk surface (i.e., between the center and the edge)
- Concurrently used datasets should be on different disks

Disk Arm Scheduling Evaluation criteria:

- Minimize mean access time
- Minimize variance of the I/O access time
- Avoid discriminations against some data placed unfavorably; avoid starvation

Main algorithms used: Scan, SSTF (shortest seek time first), cyclical Scan, others.

In some large systems, disk arms hardly move (because the disk is exclusively used for a particular kind of sequential application), so such matters don't really count. It doesn't seem that many current systems follow this pattern.

Rotational Scheduling

- Requests for different blocks in the same track can be ordered with Shortest Seek Time First. Also, a different queue may be set up for different sectors, so requests are serviced first-come-first-serve on the queue coming under the head.
- Sequential accesses can't be handled without overhead in the CPU, so skip sector allocation allows for continuous access.
- Track switching can be handled with spiraling, i.e., starting adjacent sectors at different angular positions.
- On fixed arm systems, frequently accessed blocks can be replicated at different angular positions, to be better reachable.

Prefetching On sequential access, it can increase performance by a lot. The farther ahead a sequential run is, the more likely is to jump into a different run soon. So, adjust the prefetched size according to the age of a run.

Compaction and Fragmentation

- Block Allocation
  - \- Random allocation is very bad for sequential access. No prefetching is possible
  - \+ No external fragmentation to talk about (though we still have internal fragmentation)
- Extent Allocation
  - \- Degrades over time
  - \+ It can exhibit better sequential access and it makes contiguous allocation easier.

For both schemes, compaction can be a good general solution, although it

- is slow,

- requires exclusive access to the data repository
- imposes a whole slew of updates when data structures refer to disk blocks on which a data item resides

Memory Pinning Frequently used data, especially meta-data, should be pinned in memory for faster access.

Future Directions

- Fixed block disks. They're guaranteed some minimum capacity (since we know how much the interblock gaps cost us). They're easier to search and control.
- Non-mechanical secondary storage: Powered RAM "disks"
- Smarter I/O controllers. They impose fewer interrupts and have more autonomy to plan things efficiently
- More on-disk buffering.

121 How much is enough? A risk management approach to computer security (Kevin J. Soo Hoo)

[86]

This thesis describes the need for a quantitative risk-management approach to computer security. It surveys what has been done in the past, it propose a methodology that it then validates with existing data.

Introduction & Background Information security deals primarily with four key aspects of information: availability (accessibility of the information), integrity (completenes, wholeness, and the quality of being unchanged from a baseline state), authenticity (being genuine), and confidentiality (being available only to authorized individuals). When some of these key aspects of particular information are affected, the parties interested in that information may be harmed economically or otherwise.

Traditional risk assessment describes "outcomes", that is specific ways in which information security may be hurt, and the likelihoods of these outcomes. In the 1980's, this definition of risk was used to produce a methodology for risk assessment of computer-related risk, resulting in the *ALE* (Annual Loss Expectancy) metric. ALE is the expected cost of all security-related "outcomes" given the per-outcome financial impact to an organization and the likelihood of that outcome.

Unfortunately, this metric does not distinguish between high-frequency, low-impact events and low-frequency, high-impact events, which may affect an organization in vastly different ways. As a result, a more elaborate methodology was devised, called the *common framework process*. In this process, the assets of an organization are evaluated under different security concerns (e.g., loss of integrity, disclosure, etc.). Then specific threats and vulnerabilities are identified, leading to a dearth of scenarios (combinations of asset, concern, threat and vulnerability). For each such scenario, the likelihood of occurrence and cost to the organization are evaluated. The results are then tested against requirements received from management (e.g., expected loss must be less than a given annual amount). To fulfill the requirements, appropriate safeguards are chosen. Then the process begins again, assessing the situation *with* the use of the chosen safeguards. The difference in the ALE before and after the inclusion of the safeguards can lead to a cost/benefit analysis that may improve the situation, potentially at a lower cost for the organization.

Although very flexible, the methodology failed to become particularly useful, for three primary reasons. First, its scenario orientation made it intractable to fully follow. In all but trivial environments, the number of scenarios grows exponentially with the number of choices at the asset, concern, threat and vulnerability levels, yielding thousands to hundreds of thousands of possible scenarios that must be assessed, both in terms of likelihood of occurrence and impact on the organization. Second, the methodology failed to address the realities of uncertainty in all practical systems, viewing occurrences instead as binary-outcome variables. Third, the methodology was stumped by the lack of automated or standardized tools for measuring the efficacy of safeguards or the incidence and categorization of specific occurrences. This lack of data stemmed both from an inadequate set of tools, but also from an institutional aversion towards gathering statistics, rooted in the belief that available statistics can be used against the organization in liability suits.

The miasma associated with this failed quantitative methodology threw the state of the art into the real of folk art. A "second generation" of approaches consisted of primarily qualitative methodologies. The *Integrated Business Risk-Management Framework* casts computer security breaches under the light of regular business risks. As a result, it focuses exclusively on the bottom-line business impact of general security safeguards. *Valuation-Driven Methodologies* follow up on the ALE metric, removing the likelihood of occurrences from the mix. Instead, they focus on the valuation of particular at-risk assets and propose safeguards geared towards higher-valuation assets. This can lead to oversecuring infrequently targeted assets while ignoring those that are frequently targeted but low-valuation. *Scenario Analysis Approaches* identify specific attack scenarios and propose remedies that minimize the likelihood of occurrence of those scenarios. Such approaches traditionally involve a team tasked to attack a system and produce proof of the success of the attack. Such approaches can foster a false sense of security, since they can only explore scenarios that the attacking team thought of, which may be few of many. Finally, *Best Practices* allow companies to implement those safeguards that are common in their competitive group. Although the simplest methodology to follow, Best Practices lead to an odd herd mentality where organizations constantly strive to be above the average of their group, without concern for the applicability of the specific safeguards implemented to their particular situation. Nevertheless, this is the approach most commonly taken in business.

The state of the art is likely to change soon, because of the increasing prevalence of information assets as a primary factor for the success or failure of an organization. As insurance companies enter the mix, insuring information assets, disclosure of occurrences will hopefully be forced and actuary tables formed based in part on that disclosure, that can lead to a saner, quantitative methodology for assessing security risks. Interestingly enough, "due diligence" seems to carry decreasing importance in legal battles. Judge Hand, a leading figure in the formulation of precedents in tort cases, is quoted saying "there are precautions so imperative that even their universal disregard will not excuse their omission." Judge Hand has instituted the legal use the the following rule in determining negligence: the injurer is negligent if and only if the burden or cost of adequate precautions is less than the probability of the injurious event times the gravity (impact) of the resulting injury. In other words, an organization has performed due diligence only when the cost of an increase in security/safety is greater than the benefit incurred by the

increase.

Risk Modeling and Analysis In this section, the author presents the fundamentals of risk modeling in general terms, and a few hints at how the risk model of a system can be analyzed for stability, for potential decision evaluation, and for evaluation of the value of possible future refinements.

Given a specific decision that must be made (e.g., which of 6 possible security policies should be implemented within a particular system?), modeling can take under consideration the implementation cost of each policy and the foreseen benefits from the implementation of each policy on the ALE metric. The total "net" benefit from each metric is a utility function on whose results different alternatives can be evaluated and compared to the status quo. To illustrate how one possible model can be built, the authors describe the concept of *influence diagramming*, which builds (possibly hierarchical) diagrams of deterministic or random influences on specific decisions under given utility functions.

More specifically, the net benefit from a particular policy k, in the simplest case, describes the following utility function:

$$NetBenefit_k = Benefit_k - AddedCost_k + AddedProfit_k$$

The benefit itself is described as improvement over the status quo on the ALE metric (i.e., the expected annual loss due to security breaches):

$$Benefit_k = ALE_0 - ALE_k$$

where the 0-th alternative denotes the status quo. For policies that consist of "turning on" particular subsets of fixed safeguards, this can be a straightforward determination in the simplest case. However, often the benefits from two safeguards (in terms of ALE decrease) are not additive: the sets of breaches averted due to each safeguard may not overlap (in the best case) but may also be subsets of each other (in the worst case). For increased accuracy, more refinement in the model can be helpful.

Similarly, the added profits from a particular policy translate to the sum of added profits from each individual safeguard included in that policy, although in practical settings the added profits may exhibit diminishing returns. Such is also the case for added costs due to a policy, both because parts of those costs may be amortized among multiple safeguards, but also

because some of those costs may be intangible (e.g., decrease in employee morale due to increased security requirements).

The next step, after describing a specific utility function under which different policies may be evaluated against the status quo and against each other, is to analyze and refine the model. Analysis can include range sensitivity analysis, for identifying the key variables that have the greatest influence on the decision among policies, and parametric sensitivity analysis, for identifying the variables that cause the most "cross over points" changing the ultimate decision. Stochastic analysis can further evaluate different policies when the outcome of the utility function is a probability distribution (rather than a number). Stochastic dominance chooses the dominant decision between two alternatives. First degree stochastic dominance chooses the alternative whose utility CDF is consistently lower (i.e., favors greater utility values). Second degree stochastic dominance chooses the alternative whose utility CDF *area* is consistently lower for a given utility value (i.e., the sum of all lower values times their CDF is lower, indicating lower incidence of low utility values). Third degree stochastic dominance chooses the alternative whose CDF area exhibits second degree stochastic dominance.

Of great importance also is analysis that can inform further refinement of the model, e.g., which variable to get information on. Qualitatively, sensitivity analysis can help with this concern. However, determining the *value* of additional information quantitatively can be very helpful. This is done by calculating the expected net benefit increase given a fixed value for the variable being analyzed from the net benefit measure when that variable has only a probability distribution. The variable whose expected increase is the greatest is a good candidate for further refinement since finding out more about that variable can increase the net benefit the most.

Refinement is not necessarily yielding perfect estimations for the refined variable, however. Therefore, a similar metric that calculates the expected value of *imperfect* information based on the value of the refined variable, conditioned on "further information about that variable" can help in more practical settings.

122 Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems (Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers and Douglas B. Terry)

[159]

The authors identify and shudder at the tendency towards weakly connected portable servers, which make physical and network-based attacks easier and hurt cooperative services. Portable machines are more vulnerable, even though they may be authenticated and authorized in ways similar to desktop, more secure machines. Since securing data in public systems is done often using access controls, it is serious having corrupt servers, since it is those servers that must enforce access controls. Similarly, post-mortem analysis requires an audit trail; if the host maintaining the audit trail is the potentially corrupt host, the measure doesn't really help.

The purpose of this paper is to identify weaknesses in portable-based weakly consistent systems and propose design changes to fix those weaknesses. A log-structured design is proposed that, using digital signatures, allows both on-line and off-line detection and correction of the effects of corruption. Although safety is explicitly addressed, liveness is not fully addressed here.

The model on which the paper is based consists of single-user, regular-OS computers and desktop/server computers that are physically and digitally secure. Otherwise, no network security infrastructure is assumed. Attacks, though potentially not immediately detected, are not universal; there is at least one incorruptible server. Also, a public key infrastructure allowing authentication and signing for hosts and users is required.

There is a fixed, well-known population of servers. Each server stores data modification as a log of writes. Clients submit writes to any accessible server called the *accepting server*. The accepting server assigns new writes a monotonically increasing time stamp, which along with the server's unique identifier makes up the writeID. Writes are propagated to the rest of the system via short, pairwise exchanges of updates called *anti-entropy*. Servers that pass on writes they did not originally accept are called *intermediate servers*.

The system has the *prefix property*. If I hold a write from another server, I also have all the writes that the other server has preceding the write we share. This permits the use of a *time stamp vector* as a concise digest of the state of a server: for every other server the vector has a component. Each such component holds the time stamp of the latest write accepted by the corresponding server that the local server knows about (i.e., has received directly or indirectly via anti-entropy). Write log entries that have been received by everyone may be pruned from servers' write logs. Three types of meta-data float in the system: accepted writes (conveyed in the time stamp vector), writes themselves including their writeIDs, and prunable writes (conveyed in the *acknowledgment vector* described below).

Corruption can occur in four different ways:

1. A server fabricates new writes or modifies existing writes it has accepted;

2. A server changes the meta data of writes accepted elsewhere;

3. A server modifies the meta data of its own accepted writes;

4. A server returns incorrect responses to its clients

Modified or fabricated writes are propagated via anti-entropy to all those servers who have received neither the correct version of the corresponding writes nor correct writes causing the incorrect ones to be ignored due to the prefix property.

Corruption can also change meta-data such as attribution of writes to users (which affects access controls), time stamp vectors (which can cause some writes to be discarded incorrectly), writeIDs (which can cause reordering of writes that change the image of the database). Accepting servers are much more powerful when corrupt, since there is no hope of the correct version of their accepted writes appearing somewhere by sheer luck.

The authors address these four categories of corruption incrementally. First, they address 1 & 2 on-line in restrictive model, where clients only use collocated servers to modify and query the system. Second, they address the first 3 categories in the any-client-any-server model, requiring off-line auditing for accepting server corruption. Finally, they address all four categories by pushing more responsibility onto the clients and requiring the existence of a trusted audit server.

In the first model, clients sign digitally their write requests, and server maintain those signatures. Servers sign the writeID assignments of writes they accept, also linking a new write to the write that was accepted before it. Intermediate servers cannot change writeIDs without breaking the signature scheme, though they can drop the tail of their local log. Log pruning is done via the maintenance of an *acknowledgment vector*, which, for every other server, holds the minimum time stamp that server has seen[4]. A server signs its own acknowledgment components (its own minimum-seen time stamp), so the vector cannot be advanced, only held back. The cost of this solution is constant per write (signing and extra space for tagging signatures along with the writes).

In the any-client-to-any-server model, accepting servers are audited by having clients assign unique, monotonically increasing identifiers to their own requests. Accepting servers respond to clients with a signed response that also contains the original request, thereby hash-linking responses to the corresponding requests. Clients send the responses they receive to a trusted audit server. Reordering done by the accepting server can be detected by looking at the audit trail after the fact, and walking through the hash chain of the exchanges of a client with a server. Cost again is constant per write, but a trusted auditor is needed.

Finally, to audit reads from servers, replies must include in the signed part the current time stamp vector of the replying server. This time stamp vector represents a digest of the current data state of that server. After the fact, a client can check with the audit server whether the response it got is consistent with what the data state should be, given the reported time stamp vector. The cost is very high for detecting inconsistencies.

Compared to tighter, Byzantine fault-tolerant systems, this approach is heterogeneous in the trust placed on different hosts; this inherent heterogeneity, as well as connectivity differences, make it difficult to pick a server just based on its trust history alone. Compared to Coda, this system is more complex, since Coda doesn't allow untrusted, portable machines to carry full servers, but only caches. As a result, Coda is simpler, but less flexible in the face of weak connectivity, since portables can't share new data before a strongly connected primary has had its say.

[4]This relies on the time stamp clocks of servers being roughly synchronized or share the same semantics

123 Operating System Support for Database Management (Michael Stonebraker)

[163]

Gripes about the way in which operating systems support DBMSs.

This is the earliest place I've seen the need for extensibility in filesystem caching (or as Stonebraker calls it, "Buffer Pool Management"). Problems:

- Somewhat slower
- Buffer replacement algorithm isn't controlled by the DBMS (and the DB does know exactly what it needs)
- Prefetches can only detect sequential access. Again, the DBMS knows how it should prefetch instead
- Writeback onto disk of buffers is done independently. The order and the atomicity across multiple buffers is important in transactions.

DBMSs use structured basic objects, as opposed to the unstructured character arrays (i.e., files), that normal OSes use, through their filesystems.

- Files grow one block at a time. Blocks can be distributed all over the disk's surface. Extent-based file systems are preferable.
- Two many tree structures to find an object: the one used by the indexing scheme of the DBMS, one for the file directory and one for the file blocks (through the i-nodes). One tree for all three functions would suffice.

He also argues here that an OS would be better off providing database abstractions at a lower level and implementing character abstractions on top of those, instead of the other way around (which is the current practice, pretty much).

The next gripe argues UNIX imposes the use of a DB process-per-user, as opposed to having a server that accepts messages. I don't quite see how this applies any more (he only considers pipes the only IPC mechanisms available). Should that have been the case, the problems would have been:

- Blocking for I/O leads to a large process (an entire DBMS) to be swapped, which can be costly.

- To preserve consistency, locks have to be placed on the buffer pool by competing DBMS processes. If a process gets scheduled out while holding such a lock, a convoy of waiters on that lock forms.

The alternative of having one server and multiple clients has its problems, too.

- Either the server needs to do its own multitasking, or take requests in a FIFO manner, resulting in potentially long response times (especially when, say, a long request precedes lots of short requests).
- Resources might end up underutilized, especially when there are multiple disks.
- The "have a pool of servers servicing requests" approach is very similar to having a process-per-user.
- Finally, having disk servers which service requests from the single server sounds attractive, but message overheads (at the time of writing) can be prohibitive here too.

As far as consistency is concerned, the two points apparently suggested above are:

- A specific set of buffers in the file cache has to be flushed upon commit. Control on that must be provided.
- If crash recovery is independent of transaction dependencies, then the order in which a transaction is performed might alter the results of the recovery operation after a crash.

Again, things either need to be done by the OS (if the performance issues can be resolved), or all duplicated by the DBMS.

Gripes on memory mapped files

- large sequential files (stored sequentially) have to have large page tables, which don't provide any additional information; a starting point and a length would have sufficed.
- Buffering is a problem here, too.

The size of the page table seems to have grown in orders of magnitude since then, but still exists.

In other words, OS extensibility is sorely needed.

124 Problems in supporting database transactions in an operating system transaction manager (Michael Stonebraker, Deborah DuBourdieux and William Edwards)

[164]

Yet another paper motivating OS extensibility. OSes provide transaction and recovery support, but not of the kind that Databases want. Here's the canonical version of it all (i.e., database-unspecific).

Three big issues exist:

- *Granularity* OSes support a certain granularity (page, for instance), but applications need another (record or relation, for instance). This can yield lower efficiency (especially when the discrepancy among the two granularities is great).

- *Functionality* OSes invariably provide less functionality than what applications need (except after the late '80s, early 90's, when the converse became true: more functionality was supported than anyone was ever going to need). For instance, if the OS transaction manager only offers a specific kind of locking mechanism (from the beginning of a transaction till the bitter end), but the database also needs something of the shorter-term variety, then there's a problem.

- *Extensibility* OSes define events and actions to be taken on those events. In some cases, those events and/or the associated actions are not enough; new code needs to be defined and executed with kernel privileges. In this example, the action that "reverses" an event (i.e., the undo of an event) should be customizable by applications, so that they can use their own semantics in the framework provided by the kernel. This opens up huge issues with security, trust and so on.

Solutions? Here we go:

- *Extend everything. Let applications plug in code anywhere (and deal with the slew of problems arising).* It's what the authors would love to happen. This is where the field went in the '90s.

- *Tailor applications to what the OS is offering, and do it well, too.* In some cases, it cannot be done, which further motivates the previous approach.

- *Use what the OS is offering, but make up for what's missing.* In this case (transaction and recovery of databases), it's the worst approach, since it involves a lot of complex interoperations of different modules of code. It has worked in other cases.

- *Ignore the OS and provide your own functionality.* It has been done extensively, especially by database management systems. A waste of human effort.

- *Make your functionality the OS functionality and let other applications deal.* Not quite a solution; the "other applications" would then write their own instantiation of this paper.

125 A New Page Table for 64-bit Address Spaces (Madhusudhan Talluri, Mark D. Hill and Yousef A. Khalidi)

[165]

A very interesting, dense assortment of information on page table and TLB technologies and how they scale up to larger address spaces than they were designed for.

First, different page table organizations are presented:

- Linear page tables are just huge arrays containing one entry per page. For a large address space, the page table might have to be paged, too. For something like a 64-bit address space, with 4kb pages, it might take 6 levels of paging to get to an actual word in memory. These are great in terms of locality, but the sheer size of the table can be prohibitive.

- Forward-mapped page tables split a virtual address into fields of different sizes, which are used as indices in a large n-ary tree of page table pointer entries. This is a similar approach to linear page tables, with the difference that the branching factor of the tree can be variable here,

and intermediate nodes can reside in physical memory. Intermediate nodes don't need to be entire pages, as opposed to those in multi-level linear page tables.

- Hashed page tables follow an approach close to "inverted" page tables, where the size of the search structure is comparable to the size of the physical memory, not the size of the virtual memory. A virtual address is hashed to locate a hash bucket, where the page table entry might reside. These are great for sparsely populated address spaces, but have a lot of space overhead per page table entry.

The paper introduces the concept of clustered page tables, which combine the benefits of closely-accessible neighboring pages of linear page tables and those of relatively-low overhead for sparsely populated address spaces of hashed page tables. The idea is that every node in the hashed page table corresponds to more than one PTEs (2, 4, 8, ...), for adjacent virtual pages. Therefore, within a hash node, the linear metaphor is used, but the hashed metaphor is used to locate a batch of PTEs. Clustered page tables exploit the apparent "burstiness" of address spaces, where large objects occupying more than one page are sparsely scattered in the address space. Therefore, a cluster can hold the page entries for such large objects with relatively low overhead.

The paper then goes on to present the hardware approach to larger virtual address spaces: TLBs that support superpages and sub-blocks.

- A superpage is a virtually and physically aligned block of (2, 4, 8, ...) pages, that can be accessed by a single TLB entry.

- A completely-sub-blocked page is a similarly aggregated batch of pages. Each participating page, however, can be mapped to independent physical pages, i.e., the mapping of the batch onto physical memory need not be contiguous.

- A partially-sub-blocked page is actually a superpage, some of whose individual pages might be missing. In other words, every individual page has to occupy the same page offset into the physical mapping of its batch that it occupies in its virtual mapping.

Page table support for such extended TLBs is the suggested:

- replication of extended PTEs to cover the entire area of the page table that a beefed-up TLB entry covers.

  \+ A single miss causes a mapping for the entire batch to be loaded, i.e., reduced TLB miss frequency at no cost.

  \- Page tables don't get reduced in size.

  \- Updates are more complex (the same information is stored in multiple locations).

- Multiple page tables, one per size. Search the most common one first, and on a miss move to the next most common, etc.

  \- TLB miss handlers would get much much slower, unless most misses involve the first page size.

  -/+ Page table size is reduced, but more page tables are needed.

- Linear Intermediate Nodes. Use a multiple-level tree. Internal tree nodes can be used as PTEs for a superpage the size of all PTEs this internal node represents.

  \- Only unwieldy sizes supported (e.g., 2MB, 1GB, 512GB, 256TB, 64PB).

  \+ No significant time cost in TLB miss handling for support of multiple sizes.

- Superpage-index hashing. Use the tag of the superpage and store PTEs for its components into the hash bucket.

  \- Longer hash chains mean longer access time.

  \- Superpages with sizes larger than that of the index would still need replication or some other mechanism to be handled.

Support for partial-sub-blocking is easier and more worthwhile. Using clustered page tables gives a better solution for all three extensions.

- A clustered page table could serve complete-sub-blocking directly (they use the exact same concept).

- Partial sub-blocking with the same blocking factor as the cluster is also easy to support. Instead of having an entry per base page in the hash node, a single entry with all the valid bits is used. As far as the hash table is concerned, the chain length is exactly the same.

- Superpaging is also supported, best when the only super page is the one with the same size as the cluster. Smaller superpages could also be supported, using a size field in the PTEs. However, searches could take longer, since a successful tag match with a missing mapping could also mean that a another matching tag exists down the chain, with a different page size.

126 A Survey of Active Network Research (David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, Gary J. Minden)

[166]

A high-level presentation of active network research.

Basic idea of active networks: network switches perform customized computation on the messages flowing through them.

There are two ways in which active networks have been visualized:

- *The Programmable Switch Approach.* Data passing and switch programming are discrete actions. By unbundling the programming and forwarding functionalities, we can make the programming part (which is the inherently scary, insecure one) as complex and anal as desired. For instance, complex checks can be imposed on the shipped code to make sure that it doesn't do bad things (anything from software fault isolation to instrumentation, as in vino).

- *The capsule approach.* Data packets are replaced by data/processing capsules, much like in the case of postscript replacing a text document.

126.0.5 Programming Model

Program encoding must, in general, support *mobility* (i.e., be independent of platform and location), *safety* (i.e., provide ways to restrict the accessible system resources), and *efficiency* (i.e., make sure that in most common cases performance is not horribly penalized).

The kind of functionality available would cover packet manipulation, access to switch environment (time, address, link status, etc.), packet flow manipulation.

The program model must also defined a common resource model (routing tables, link status, routing capabilities). Defining a secure resource allocation mechanism is a hard problem.

Systems with use in this framework include Safe-Tcl, Java, Omniware and Proof-carrying code.

126.0.6 Advantages – Proposed Uses

- New protocol deployment could turn out to be instantaneous.
- User-aware network protection. Use exactly as much as you need. Not entirely clear how you'd do that.

126.0.7 Disadvantages

- Infrastructure safety. Unleashing arbitrary pieces of code in core switches can be hazardous to the network's health.

126.0.8 Commentary

Some additional issues/disadvantages not clearly mentioned are

- State tracking: Whether soft or congealed, state has to be maintained all along a network path, which means more memory, operations to retrieve it and so on. To make the state soft, packets have to have redundant information to potentially regenerate expired state, or latency might have to be added for on-demand state retrieval. In the case of hard state, we have to deal with robustness, invalidations and the like.

127 Session Guarantees for Weakly Consistent Replicated Data (Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, Brent B. Welch)

[171]

This paper introduces the concept of *session guarantees* in the context of a replicated database system. A session is an abstraction for the sequence of read and write operations performed during the execution of an application. An application can request a certain guarantee from its session, ensuring that certain properties are true during the exchanges of the application with replicas of the database.

The key point here is that the application can request and receive guarantees from its own session that do not apply to all other sessions of all other applications using the same replicated database at the same time. In a sense, the application receives selfishly a "consistent" view of the world, despite the fact that others—who have not requested any guarantees—might be seeing world views that are inconsistent.

For example, if I require that I see a reply to posted message in a USENET system only after I've seen the original message, then I can request a pertinent guarantee from my newsreader. Other users of USENET may still get messages out of order, perhaps faster than I will, but I will be guaranteed to get a view of USENET that is consistent with my requirements.

The authors define four sessions guarantees:

1. *Read Your Writes.* If I write something into a first replica, then subsequently I can only read something from replicas that contain my earlier write. This guarantee does not allow me to see a view of the database that does not include my earlier writes.

2. *Monotonic Reads.* If I read a bunch of data from a first replica, then subsequently I can only read from replicas that contain have incorporated the writes whose results were read by my first read. This guarantee prevents me from seeing a state of the database that is less up-to-date than what I have previously seen.

3. *Writes Follow Reads.* If I read something from a first replica and subsequently I want to write something back into the database, I can only write data into a replica that contains all the writes whose results were read by my earlier read. Futhermore, this restriction is applied even when my write is propagated to other replicas in the background; my write will never be seen in a replica that does not hold the writes on which my earlier read relied.

   Similarly to monotonic reads, this prevents me from writing into a state of the database that is less up-to-date than what I have previously seen. Furthermore, it ensures that my writes appear only in replicas that hold the state that "influenced" my writes.

4. *Monotonic Writes.* If I write something into a first replica, then I can only subsequently write into another replica that already contains my previous write. As in Writes Follow Reads, the same restriction is applied when my later write is propagated to other replicas in the background.

   This makes sure that the ordering I have imposed on my own writes is enforced anywhere my writes appear.

The authors have built an efficient implementation that allows any combination of the four session guarantees to be enabled within a session. Their implementation enforces slightly stronger guarantees. However, this conservative approach is justified by the simplicity of the approach while retaining the importance and flexibility of the guarantees.

The implementation relies on the concept of the *version vector*. A version vector characterizes the state in which a particular replica is with regards to the foreign writes (i.e., writes performed elsewhere) that it has received and applied locally. Each replica utilizes a logical clock that it increments every time it receives a write from a client. A replica assigns its current clock value to a write it receives from a client.

The version vector of a particular replica contains an element for each replica in the system. The element for the local replica contains the current clock value. The element for a remote replica contains the clock value of the latest write from that replica that has been incorporated in the local database. For example, in a system with replicas A, B, C and D, is the version vector of replica C is $[5, 9, 13, 4]$, then the replica has seen the first 5 of A's writes, the first 9 of B's writes and the first 4 of D's writes. Furthermore,

C's current clock shows 13. When replica A propagates database updates to replica B, it only propagates writes performed at C if A's version vector has a higher third element (i.e., element for C) than B's version vector, and propagates writes starting from the earliest going up in order.

To maintain session guarantees, a session maintains two version vectors for itself, a read vector and a write vector. Informally, the read vector shows the "state" of the database as seen at the replica where the session performed its latest read. The write vector shows the state of the database as seen at the replica where the session performed its latest write.

During reads, a replica returns its current version vector along with the result. The session replaces its own read version vector with the vector whose elements contains the maximum values at each elements of the replica's version vector and its own read version vector. During writes, a replica returns its current clock to the session. The session replaces its element for the replica at which the write was performed with that clock value.

Read Your Writes can be ensured during a read by checking that all elements at the version vector of a replica are greater than or equal to all elements of the session's write vector. That is, all the writes that the session has experienced at its last write must have been propagated to the replica before the session reads from it.

Monotonic Reads can be ensured during a read by checking that all elements at the version vector of a replica are greater than or equal to all elements of the session's read vector, similarly to Read Your Writes. Now, the replica contains all writes that the session's previous read experienced.

In equivalent terms, Writes Follow Reads and Monotonic Writes are ensured during a write by checking that all version vector elements at a replica are greater than or equal to all elements of the session's read vector and write vector respectively.

Version vector checks need only be performed when a session switches replicas.

128 Towards a Quality of Service Model for Replicated Data Access (Douglas B. Terry)

[170]

This short position paper draws an analogy between the concept of Qual-

ity of Service in networking and quality of service in a distributed database.

Replicated databases or file systems may present different consistency models to their clients: a strong consistency model, such as one-copy serialization, or weak consistency with eventual convergence (e.g., Locus [176] or Coda [97]. Strong consistency is not always desirable, since it causes low performance in general, and is not always feasible, as in the case of mobile systems.

The author claims that clients of a replicated database should be allowed to specify their "QoS" requirements from the system. Such requirements could be the session guarantees from Bayou [171], data freshness or even strong consistency.

129 Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System (Douglas B. Terry and Marvin M. Theimer and Karin Petersen and Alan J. Demers and Mike J. Spreitzer and Carl H. Hauser)

[172]

This paper describes the conflict resolution machinery of Bayou, a weakly consistent replicated storage system.

The main selling points for the system built in this work are the following:

- It requires only occasional, short-lived connectivity.
- It supports disconnected work-groups, i.e., doing useful work even while disconnected from the rest of the world.
- It is no intended for transparent replication; instead, applications are required to participate in conflict resolution.
- It offers fine conflict resolution granularity: resolution is done on a per-update basis.
- Tentative data, that is data resulting from writes that have not yet committed, are available to client, albeit marked explicitly as such.

The paper contains the system's mechanisms for providing dependency checks and merger procedures per update, for differentiating tentative from committed data, for doing access control, and, briefly, for achieving eventual consistency.

Bayou is based on the read any/write any data model, whereby a client may interact with any server, whether or not the global system is reachable at the time. It uses session guarantees [171] to allow different applications to get a view of the global system that is consistent with their own actions. Servers use pairwise communication, called *anti-entropy*, to exchange and propagate updates, so as to attain eventual consistency.

Bayou has been validated against two applications: a meeting room scheduler and a bibliographic database. In the meeting room scheduler, many users schedule meetings to take place in specific rooms. A user may tentatively schedule a meeting while disconnected, or only connected to a small disconnected work-group. Tentatively scheduled meetings are marked as such in the UI. To deal with conflicts, a room reservation is accompanied with alternatives, which can be tried in order if the first choice ends up being unavailable during conflict resolution.

The bibliographic database maintains a list of bibliographic entries indexed by a unique key, consisting of an author last name and the year of publication of the entry. Name conflicts can arise when multiple entries by the same author and year are entered. Similarly, duplicate entries may be inserted by multiple clients. The system resolves name conflicts by creating slightly modified keys for all but one of the conflicting entries. It solves duplication by, usually, merging duplicate entries together into one.

In Bayou, every update, besides its update operation (SQL-based), it contains a dependency check (also an SQL SELECT) and a merge procedure (a TCL code fragment). When the update operation reaches a server during anti-entropy, the server runs the dependency check. If the check succeeds, then the update operation is run. Otherwise, the merge procedure is run, to fix the conflict.

Updates start their lives as *tentative*, i.e., not guarantee to stick. Servers make the results of such updates available to their clients; they mark them explicitly as tentative, however, and it is up to the application to decide whether to use or not tentative data. Tentative updates are retained in a server in a undo/redo log, so as to deal with out-of-order delivery of disparate updates.

An update is eventually committed. A single server serves as a primary

for the commit operations of all updates involving a single data set. Once the primary receives an update and accepts it (with or without conflict resolution), the update is marked as committed, and propagated back into the system. The ordering of events produced at the same server is not always the same as the ordering imposed by the associated commit server. The former is determined by how the events are submitted to the originating server. The latter is determined by how the events reach the commit primary server. The latter ordering is the one that persists.

The undoability/redoability of merge procedures is preserved by enforcing that such procedures only use stored data, not local environment information such as the local time, the local hostname, etc. As a result, a merge procedure always has a deterministic outcome when applied to different servers who have, however, seen exactly the same updates.

The per-update dependency check/merge procedure scheme is more expensive than having per data-type conflict resolution machinery, but is much more flexible. In practice, Bayou merge procedure functionality can be factored into library functions that many merge procedures share, thereby reducing the cost. In a sense, this emulates a data-type-specific conflict resolution scheme, but still allows the more flexible per-update scheme too.

Every system operation is authenticated and authorized with a certificate. Privilege certificates are signed by a single, globally trusted authority, and associates a public key with a privilege on a particular data collection (note that granularity here is that of the data collection). Privileges can be delegated using revocable certificates signed by the delegator. Revocation of both privilege and delegation certificates is done using revocation certificates, signed by the originator of the revoked certificate.

Certificates are checked between clients and servers, when an operation on the database data is attempted, between servers during anti-entropy, and at the commit server, during update commitment.

130 Dealing with Tentative Data Values in Disconnected Work Groups (Marvin M. Theimer and Alan J. Demers and Karin Petersen and Mike J. Spreitzer and Douglas B. Terry and Brent B. Welch)

[173]

This position paper touches on the concept of tentative writes in a weakly consistent replicated database system, where groups of servers become disconnected from the rest as a matter of course, not exceptionally. In such a context, the users of the disconnected replicas of the database invariably have to read data values that have not reached the entire system and that may very well conflict data values created at a different partition of the system. Worse yet, the clients in a partition may rely on such tentative data to create further writes into the same or even other databases.

The approach that the authors advocate in the context of the Bayou project consists of four requirements:

- Keep track of which client has seen each tentative update during disconnection.
- Keep track of dependences, implicit or explicit, between a tentative update value and other actions in the system.
- Notify clients of a discovered conflict that involves them directly or indirectly, allowing them to take reconciliatory actions.
- Make explicit the state of a data value (tentative or committed) at the time when a client accesses it. A smart client will only perform undoable operations in response to a tentative data value.

131 Active Names: Flexible Location and Transport of Wide-Area Resources (Amin Vahdat, Michael Dahlin, Thomas Anderson, Amit Aggarwal)

[174]

Another take on the deployment of wide-area services.

The spin is that instead of plugging the code where the data are going to pass through (which is what Active Networks do [166]), we map the path name to a sequence of operators (i.e., we let the path be calculated dynamically).

This is an overlay mechanism, in that IP-level routing is not supplanted.

Example applications would be replica selection in a replicated framework, mobile distillation (where the distillers can move to minimize perceived latency), or customizable client caches.

This is yet another approach to naming things using the "what", not the "where", otherwise called *Intentional Naming* [1] or anycasting [134]. Intentional Naming bundles location and transport in one nice package, which allows location to be swayed by available transports or, conversely, transport be swayed by how we named the endpoint. In contrast, "Extensional Naming" breaks up location and transport. This might seem simpler (since it is conducive to modularized solutions), but it implies some hard state: If a name resolves to a transport address, this address must be continuously supported, though it might have nothing to do with what is named (e.g., in mobile systems).

Active names pay the mobile code premium once per session, whereas active networks pay it per packet. This allows active networks to change their behavior per packet, which active names cannot.

131.0.9 Design

Their main design goals are:

- Namespace customization and extensibility. The system must be "complete", in that it should be able to do anything (?).
- composability

- efficiency
- location-independent resolution

The scheme works like this:

1. Client C wants service S.
2. C builds an Active Name AN: [Name N:Namespace Program NS].
3. AN is shipped to the nearest resolver R.
4. R locates NS from the reference to it in AN.
5. R runs NS on N. It resolves part of N, which it uses to figure out where (what namespace program NS') to send the remainder N' of N. The computation it determines that has to be performed is formulated as an *After-Method* AM and pushed down a stack in AN.
6. N' is shipped to another resolver, where NS' is going to be executed to resolve it and so on, until a leaf is reached. Along with N' and NS', the stack of after methods is also moved along.
7. When the entire name has been resolved, a chain of after methods has been derived. After methods are bidirectional pipes, chained together into one pipeline where the input data (S) is going to be directed. The output of the pipeline will be delivered to the destination (customarily, C).

In 4, the location of NS may also be an active name. Obviously, there needs to be a ground case in this recursion. Such is the case for some primitive services (like HTTP or DNS).

Resolvers are pretty much compute servers with some necessary functionality:

- They must be able to fetch names and namespace programs
- They must be able to execute programs safely in a controlled environment (something like a sandbox virtual machine).
- They must be able to retain (cache) soft state about previous resolutions.

- They must be able to communicate with next-hop resolvers.
- They must be able to respect authentication/authorization restrictions imposed by namespaces.

131.0.10 Commentary

The paper compares active naming to active networks or active services before it has even talked about what exactly active naming is (they do talk about it, but in too fluffy terms to actually be helpful). This can be very annoying.

Unfortunately, according to the authors, there are many security issues that still need to be resolved.

The described applications of the scheme are pretty convincing, especially in the case of mobile distillation.

132 The LOCUS Distributed Operating System (Bruce Walker, Gerald Popek, Robert English, Charles Kline, Greg Thiel)

[176]

LOCUS was a distributed operating system running on heterogeneous machines. It had a distributed, replicated file system, supported nested transactions, provided location-transparent naming and so on and so forth.

Replication in a file system is great since it enhances availability and performance. However, it can be hellish for maintaining concurrency control.

There are multiple physical devices per filesystem (since its replicated). Normally, physical devices are incomplete (i.e., they don't contain the entire logical file system). Version numbers are used to maintain concurrency. Access is transparent, regardless of the physical location of a file (except for differences in performance).

The synchronization site (CSS) (which is the synchronization manager for a file system) is assigned for all files within a filesystem. Updates are possible even when not all copies are accessible (i.e., when the network is partitioned). However, upon reintegration of the network partitions, a reconciliation mechanism is applied to bring all copies of the file in synch, especially if all disconnected copies had been updated in parallel.

The CSS is maintained in the mount table, which is globally replicated. Subsequent accesses to a file (after the CSS has been consulted) don't have to involve the manager.

Directory path traversals are done a component at a time, but shipping multiple components to a remote storage server for a directory was considered.

File operations are atomic. Commit and abort are supported. Old data versions are kept through shadow paging. Update propagation is also done in an atomic way, in case contact is lost halfway through the update.

File creations and deletions are handled just like file modifications. Storage Servers (SS) for new files are chosen among those of the parent directory.

For architecture-specific file location (different executables for different architectures)), a hidden directory is created with the name of the ambiguous file. Inside this directory, files named after different architectures are contained (e.g., /bin/ls/i386 is the file that's going to be executed when I execute /bin/ls on an i386 machine. On a SPARC machine, /bin/ls/SPARC is going to be executed instead).

Processes can be forked locally or remotely in a transparent way. A combined fork and exec has been provided, to avoid futile address space copying (especially over a network link). Shared resource semantics (open files, pipes) is handled through a token technique, whereby access is only allowed when the token is held by the node.

Within a partition, shared resources are strictly synchronized. Upon reintegration, conflicts are resolved automatically for understood resource types (say mailboxes), or a higher arviter is required. Partitions aren't just disconnections through router failure; they can be due to maintenance, negligence and so on.

Primary copy, majority consensus and weighted voting can allow updates on only one of the partitions.

Both bundled updates and unrelated updates are handled.

Reconfiguration involves:

Partition detection Find maximum partitions visible from each of their members.

Merging Attempt to merge the current partition with other partitions that might have reconnected since the last detection phase. Also reset CSSes and mount tables.

Cleanup After a change in the partition membership, reset locks and such to reflect current conditions.

The three reconfiguration protocols are combined, preventing deadlocks by using protocol ordering (i.e., being strict on what runs when) and node ordering.

133 SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Matt Welsh, David Culler and Eric Brewer)

[177]

This paper goes back to a classic old religious battle in computer science: monitors vs. processes. One model structured applications as a single thread of control that multiplexes itself among several tasks, by picking up events from an event queue and executing them in order. The other structured applications as a pool of threads of control, one for each task; the more concurrent tasks there are, the more threads of control exist advancing the collective progress of all tasks one quota at a time. The classic Lauer and Needham paper [108] settled the score, by showing that the two models are duals of each other. In this work, the authors revisit the problem and show that combining the two model, building a hybrid that takes advantage of both models, can yield significantly favorable results.

The hybrid architecture, SEDA, breaks up an application into *stages*, much like the stages of a hardware pipeline. Stages communicate with each other with events (messages) that are placed in per-stage event queues. Each stage manages its own thread pool, based on a local, per-stage policy, with the sole purpose of serving its own outstanding events in its queue.

The benefits of this architecture lie in the decoupling of vastly different portions of a task—CPU-bound vs. I/O-bound—without depending on a large number of concurrent threads, since many threads means many context switches. Each task can increase or decrease the size of its own thread pool based on its own performance requirement (service rate, input queue length, etc.).

In the process, the authors have also constructed an asynchronous socket and file input/output library, for use with their SEDA-based framework.

The validity of the architecture is exhibited with two applications built using it: a web server, and a file-sharing client. Especially for their web server application, the results are astonishing: the server, built on Java, outperforms the industry standard Apache server, built on C, and fine tuned over long periods of time, on throughput, response delay variance and fairness.

This architecture seems very promising, although it is not clear whether all or most performance-critical applications can be broken down into self-contained stages, in the manner of web- or file-servers, or if applications with many different stages can actually benefit.

134 Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0 (Alma Whitten and J.D. Tygar)

[178]

A very unique paper on the usability of a popular security application (PGP 5.0 for the Mac).

The main point raised here is that it is not enough to build security applications with regular user interface design principles in mind; Security applications have much more specific usability goals, which cannot be met by commonly used design principles.

The authors define security software as usable if "the people who are expected to use it:

- are reliably made aware of the security tasks they need to perform;
- are able to figure out how to successfully perform those tasks;
- don't make dangerous errors;
- are sufficiently comfortable with the interface to continue using it

They evaluated PGP 5.0 for the Mac, as used through the Eudora mailer, against this definition of usability, following two complementary techniques: first, they performed a cognitive walkthrough, an inspection of the user interface by usability experts; second, they ran a user test on 12 subjects who had no prior experience with security applications.

Their conclusions were grim for the software, although PGP 5.0 exhibits a solid design according to general design interface rules. The application displayed lacked a consistent, comprehensible presentation for the security model involved (concepts like public/private keys, web of trust in key signers, public key servers, decryption vs. signature verification, etc.), which led users to perform substandard secure tasks.

In conclusion, the authors encourage the development of a body of work on usability evaluation in a security context.

135 Fremont: A System for Discovering Network Characteristics and Problems (Wood, David C. M. and Coleman, Sean S. and Schwartz, Michael F.)

[179]

The basic idea is the parallel exploitation of multiple sources of network information towards the same goal. A journal is created using information drawn from ICMP, ARP, SNMP(future work), DNS, and RIP. Although each method individually has results with limited coverage, overlapping the results from multiple methods can cover the gaps.

Most techniques are fairly intuitive (pinging address ranges, broadcasting pings, exploding TTLs as in traceroute, DNS zone transfers, ARPing subnet ranges, snooping for ARP replies, etc.).

136 Recommendation X.500: Data Networks and Open System Communications - Directory

[91]

The ITU-T take on the Directory service.

136.0.11 Basic Assumptions

A basic objective is to provide "user-friendly naming", whereby named objects have names suitable for citing by human users. This is not all-restrictive, i.e., some names might also be "unfriendly".

The other basic objective, naturally, is to contain "name-to-address mappings" for indexed objects which are dynamic. In other words, the served mappings may change over time.

This is not a database service. Queries are assumed to be much more frequent than updates. Rates of change are assumed to follow "human standards" as opposed to "computer standards". In other words, a mapping is assumed to change at rates no faster than those that a human could accomplish.

Eventual consistency is what counts here. Cases where both old and new versions of some information float transiently through the system are acceptable.

Except for access rights issues, results of a query are independent of the identity or location of the inquirer.

The data model is assumed to be structured as a tree (and the interfaces take that under consideration). However, other structures are not precluded from existing in the future.

136.0.12 The data model

The data model (as described in X.501, which is as yet not electronically available), is hierarchical, based on value-attribute nodes. A node normally follows a schema that defines its type (*class*). Administratively, the party responsible for a node defines its own class and nothing else. Of course, defining the class of a node also defines the possible classes of the node's descendants.

Aliasing is allowed. An alias node may point at a node somewhere else in the data hierarchy.

Nodes (i.e., entries in the directory), have distinguished, unambiguous names which are derived from the path connecting them to the root of the hierarchy.

Intermediate and leaf nodes in the entry tree are identical in structure (i.e., there's no special leaf node type).

136.0.13 The Directory

The directory provides support for querying and modifying it. It provides responses to all types of requests. Responses are however specific to the request type.

The Directory offers optional support for mutual authentication of the user and the directory.

It is possible to qualify how a request is going to be handled:

- Controls exist for limiting resource usage, interaction modes and so on.
- A request may be accompanied by security info (digital signatures, for example).
- A request may be accompanied by a limiting filter that helps return a better selection of results from a larger class of them.

Query requests include read, compare (for instance, for password checking), list, search and abandon (which tells the directory a previous request is no longer relevant or needed). Query requests may contain a context, that can be used when multiple versions of values are kept for some of the attributes of a sought entry. A good example of context is "language".

Modification requests include add (leaf), remove (leaf), modify entry (which is an atomic operation) and move (which moves an entire subtree into a new location). Contexts may also be included in the request.

Exceptional conditions include errors (for anything from security issues to schema rule violations or service controls) and referrals (which allow a directory access point to refer a user agent to another, more suitable access point – perhaps because the latter is closer to the requested data).

Requests are handled by directory service agents either recursively (in which case, if they don't personally know the answer, they'll find it from other agents and return it to the requester), or by referral (in which case, they'll point the requester to a more likely place for the posed request).

136.0.14 Distribution Issues

The Directory service supports redundancy, in the form of shadowing (as defined in X.525) or caching (which is considered an implementation issue, as yet unspecified).

The shadowing model assumes a single owner of specific data. Shadowing is read-only. All modification requests are forwarded to the data owner. Partial shadowing is also permitted (whereby, not all components within a subtree are shadowed; a filter predicate is used to define which parts of a subtree are to be shadowed).

Inconsistencies are tolerated. In the case of shadowing, inconsistencies are always temporary (although no time constraints are specified). In the case of caching, no declaration is made. The only stipulation mentioned for shadowing inconsistencies is that they never occur internally to an object, i.e., the fields of a single object all contain information from the same version of that object, whereas a shadowed copy of that object in a different directory agent might contain an (entire) previous version of it.

Replication is not transparent to the user. The user knows if a request was satisfied from a shadow copy of the authoritative data. The user is also allowed to specify that a request should be satisfied by the master directory server, for better accuracy, at the expense of performance.

136.0.15 Access

Access to the Directory is allowed through the *Directory Access Protocol*. LDAP is a light-weight version of that protocol.

136.0.16 Commentary

It is not clear how to deal with transient inconsistencies, when, for example, both old and new information floats on the system. Is there an upper bound to the duration of the transient state? How can we (eventually) tell the information we received was about to be purged from the system?

It's not clear what kind of filtering they intended to support in requests, as shown in 8.2.3.

It's not clear what the use of a context in add or remove requests might be, in 8.4.

It would be interesting and useful to allow aliasing inside an object (i.e., to give different names to the same attribute inside an object).

137 Recommendation X.501: Data Networks and Open System Communications - Directory: Models (ITU-T)

[92]

This standard describes exactly how things are modelled inside X.500 ([91]).

An administrative domain, as far as X.501 is concerned, consists of multiple subtrees not necessarily communicating with each other. This means that the domain may control multiple subtrees routed at different points of the global tree.

X.501 seems to have a strange fascination with administrations of the national level. They even make the distinction between the domain belonging to a national "administration" (and administration is the national telecommunications carrier) and private administrations. They even go so far as mentioning national regulations governing administrations within the barriers of a single country.

The specification recognizes the existence of multiple roles of a single person. They don't go far enough to define a correspondence between real objects (i.e. the real world) and online objects.

The administration of the directory is served by entries in the directory itself. Although those entries are mentioned as "administrative entries", they look exactly the way other normal entries look.

Attributes types may be subtyped. Every entry in the directory, contains an attribute type and a list of values (non empty). A value may be annotated with a context. A particular entry may have many alternative values for various contexts. A common context is language.

Additional object schemas may be defined, beyond those standardized by central organizations, national organizations or private organizations. It is not clear how easy to do that will be.

Besides attribute types, there are also object types. Those are also subtyped. Multiple inheritance is supported.

There is a fine distinction between what is called a structural object class and an auxiliary object class. Roughly speaking, the former is an interface whereas the latter is an implementation. Structural object classes represent the way an entry looks inside the directory. An associated auxiliary class determines the semantics of that entry.

In section 8.8, matching rules are discussed which are used for searching through the directory. Although a set of basic matching rules is defined in X. 520, no restrictions are placed on the kind of rules supported in X.500. The most basic rule is an attribute-value assertion. This checks an entry for the presence of an attribute and compares its value with that given in the assertion. Such rules may have a context associated with them, so that a particular instance of the sought attribute is checked. Every attribute type contains its own equality matching rule (somewhat like the equals method in Java). That can be used in attribute-value assertions. Rule evaluation is extremely flexible, to allow the same rule to match different things depending on where the directory resides. For example, if no context preference has been included in an assertion, then the context may default to what the particular answering directory considers default. Builtin rules include presence, equality, ordering, substring matching, and approximate matching.

Section 9: A name is a sequence of Relative Distinguished Names. X. 501 acknowledges that alternate ways to name objects in the directory are possible in the future. An RDN is the set of attribute value pairs in an entry of all those attributes which have a distinguished value. If an attribute doesn't have a distinguished value then it doesn't participate in the RDN of its enclosing entry. An entry may have multiple RDNs if some of its distinguished values has instances for different contexts. An RDN is modifiable.

Among the multiple RDNs for an entry, one is designated the "primary" RDN for it. It consists of the primary attribute value pairs for all participating distinguished attribute value pairs. This primary RDN, which is unique for an entry, is what most directory operations would normally return.

A distinguished name for an entry is the sequence of RDNs from the root of the directory to the entry, through the hierarchy. A distinguished name for an alias is not the same as the distinguished name for the entry the alias points to. Clearly, a primary distinguished name for an object is that consisting of primary RDNs.

Alias entries are always leaf nodes. They do not have to point to the name of a real object necessarily.

Section 10: Administered areas are partitions of the global tree that are administered by a single administrative organization. For example, the subtree belonging to the U.S. constitutes an administrative area administered by the actual organization in the U.S. dealing with its share of the X.500 global hierarchy. The root entry of such a subtree is the administrative point.

Inside an autonomous administrative area, there are four kinds of specific

administration supported: for names(i.e., deciding which names can or cannot be given), schemas (deciding how governed entries will be structured), security and collective data. Specific administrative areas deal with subtree schemas, security (i.e., access control), collective entries and default contexts (i.e., which context will be considered the default one, when the default context is needed).

Areas dealing with specific kinds of administration among those above may overlap; for example an administrative area that deals with security might overlap with a different administrative area that deals with the default context. A domain as described above consists of potentially multiple autonomous administrative areas.

Section 11: A subtree refinement is defined as a subtree, whose shape has been carved either by depth (i.e., excluding a number of upper levels and or lower levels), by name (i.e., excluding a number of its subtrees with or without the subtree root) or by type (i.e., excluding or limiting to a set of entry types).

Section 12: A directory schema, associated with an administrative area, contains information about the kinds of names possible in the area, the possible object classes, attribute types, matching rules, contexts, associations of attributes to possibly containing object classes, and possible associations of object classes with subordinate object classes in the hierarchy.

Class hierarchy is explicit in the directory. More specifically, every object contains the identifier of all the superclasses of its own class. This need not necessarily be reflected in the actual storage. The directory does make sure that entries of a particular object class follow the rules (contain all the mandatory attributes, do not contain unanticipated attributes, do not allow multiple values in singleton attributes and so on).

Attribute types contain information such as their superclass, their syntax, matching rules, whether they are operational (i.e., meta-informational) or user controlled, whether they are collective or not and whether they are singleton or not. Operational and user attributes form separate hierarchies. Attributes lacking equality matching rules cannot be used for naming and lose their type when queried.

To fully specify the structure of an administrative area, its administrative point will contain enough subentries describing allowed structural classes, their naming and their content rules. Name forms are assigned to an object class and determine which of the mandatory or optional attributes of that class can be used in its RDN.

Structure rules specify how entries can be linked to each other in the hierarchy within an administrative area. They associate a name form (which, in turn refers to an object class) with a set of possible superior structure rules (which, each, point to object classes as well). For example, to make it possible for an entry of object class A to have a subordinate entry of object class B, first a name form for class A has to be created, let's call it name form C, that defines an RDN scheme for class A, and then a structure rule D for name form C (connecting it to its superiors). Then, a name form E for class B should exist and a structural rule F containing E as its name form and designating structural rule D as its superior rule.

The actual contents of an entry within the specified administrative area are dictated by a content rule. A content rule takes a particular object class and turns it into a specialized/customized class for this administrative area. It can give it additional mandatory and optional attributes (beyond those defined in the class definition), specify additional auxiliary object classes ("implementation" classes), and preclude some of the inherited optional attributes from appearing.

Finally, contexts within the area are defined in two phases: context types are primary, fundamental contexts that can be used within the area. Those are defined independently of the attributes where they may apply. In the second phase, context use rules associate an attribute with a set of mandatory contexts (they all have to be defined whenever this attribute is met in the area) and a set of optional contexts.

Section 13: The directory system schema is defined within an area to determine how administrative and operational properties of area can be included. It determines the kinds of subentries that may be created and defines operational attributes for the area. So it deals with subentry name forms. Other operational attributes include timestamps, creator or actor identities, associations of entries with their governing subentries, access control, context assertions.

Section 14: In this section the actual structure of a subentry is described and how all pieces of schema information from sections 12 and 13 come together into the subentry of an administrative point.

Section 15: Authentication procedures work through the association of a user identity and an access control identity. The latter is used in all access control decisions.

An access controlled specific administrative area is governed by the access control scheme defined in a subentry of its administrative point.

All arguments, results or errors in directory operations may use a protection mapping. There are three defined protection mappings: signing, encryption, and signing-and-encryption.

Section 16: Several access control schemes can be utilized inside the directory, to suit the requirements of the current administrative area. Basic access control is fairly flexible and somewhat complicated. Simple access control is a subset of basic access control. Rule-based access control can setup procedural kinds of access decision functions.

In access control schemes, access control items are the basic units of control specification. They can be placed at various locations throughout an administrative area, with different semantics. Sets of them can be combined and precedence is defined among access control items in a set according to an integer priority. An item can have different levels of fidelity requirements placed on the authentication of the requester of an operation. Specifically, the identity of the requester might have no fidelity (i.e., we assume the requester is who it claims to be), simple fidelity (i.e., password based) or strong fidelity (i.e., based on strong authentication).

Access control can apply to a number of different kinds of "protected items". Such are entire entries, attributes, attribute values or names. More specific instances of those may be specified as protected items, such as "all attribute values of attribute X". additionally, access controls can be associated with constraints, such as "do X if by doing so there won't be more than y values for attribute z".

Permission categories include standard permissions like read, browse, add, modify, permissions for moving entire subtrees to a different location, for using attributes or attribute values in filter matches etc.

Permission items can apply to an entire administrative area (in which case they are called prescriptive and are placed within a subentry of the administrative point), to the subentries of an administrative point (in which case they are called subentry and are placed within an administrative point), or to a single entry (in which case they are placed within the entry itself). When making an access control decision, all access control items of the scope within which the protected item resides are taken under consideration. Those are the entry item, if one exists, and all prescriptive items for all administrative areas within which the item lives, up to the top of an autonomous area. Similarly, for a subentry, its own entry item, its owning administrative point subentry item and all superior prescriptive items up to the top of an autonomous area are taken under consideration.

A general rule governing how basic access control schemes operate is that any access control decision should be answerable without having to inquire remote directory system agents.

The "simplified access control" scheme is the same as the "basic access control scheme", except that it doesn't support inner security areas (i.e., no nesting) and it doesn't support access control items within single entries.

Section 17: In rule-based access control schemes, access control associated with specific data is not explicit. Instead, a general label indicating how restricted a piece of information is is attached to the relevant entry. This label along with the properties of the requester and the policy used determine whether access will be granted or not.

Section 18: X.500 provides support for maintaining integrity information within the directory. This is done by supporting digital signatures within the directory along with the information itself.

Similarly, X.500 provides support for maintaining confidentiality of the stored information (i.e., keeping stored information secret even from the directory itself). This is done by designating stored information as encrypted using a particular algorithm.

138 Analyzing the Secure Overlay Services Architecture under Intelligent DDoS Attacks (D. Xuan, S. Chellappan, X. Wang and S. Wang)

[180]

This paper expands on SOS [96], describing a generalization of the beacon concept to multiple beacon layers, and pointing out effective adaptive attacks against the original SOS design, along with possible defenses for them.

The basic "intelligence" in the attack strategies proposed by the authors is to divide attacks into a break-in phase and a congestion or flooding phase. In the break-in phase, the adversary spends a fixed budget of break-in "chips" on taking over overlay nodes. Using these nodes, the adversary can locate where nodes (beacons) of the next stage lie; in the original single-beacon-layer SOS architecture, break-ins into the SOAP layer allows the identification of beacons, who hold the identities of secure servlets.

In the second phase, the adversary congests by flooding or otherwise sensitive overlay nodes (beacons at different beacon layers), so as to remove as many possible paths from clients to the target end-point as possible (combining broken-in nodes and congested nodes).

The authors conclude by defining a "design problem" of obtaining optimal sizes of each layer and "connectivity" from layer to layer, so as to maximize the probability that some paths from clients to the protected target remain operational.

139 Design and Evaluation of a Continuous Consistency Model for Replicated Services (Yu, Haifeng and Vahdat, Amin)

[181]

This paper presents one of the first (if not the first) treatment of consistency as a continuous system design parameter. It describes how consistency can be broken into individual metrics, which can in turn be made as expressive as needed to cover most traditional consistency models: single-copy serializability and optimistic consistency.

Consistency is broken into three different metrics: *Numerical Error*, *Order Error* and *Staleness*.

Numerical Error This bounds the discrepancy between the value of a data unit and the "final image" of this data unit, i.e., the value of that data unit in the fully up-to-date version of the repository. For example, this would be 0 for a replica, if we are unwilling to tolerate the least inconsistency, ever. On the other hand, a value of 10 for this metric would mean that we are willing to tolerate inconsistencies that will keep the value of our cached copy within 10 of the "final image" of the data unit.

Order Error This measures the difference between the order in which updates are applied to a replica, relative to their ordering in the "final image". Order error of 0 means that no updates should ever have to be rolled over so that they can be reapplied in the correct order. In other words, writes must be synchronous for all replicas. On the other hand, an order error value of 5 means that at most 5 updates may be applied

to a replica *tentatively*, even before it is known that those updates were applied in the right order.

Staleness This measures the difference between the current (logical) time and that at which the oldest write on a data unit was seen at a replica. Staleness of 0 would mean that a replica would have to always verify that it holds the latest version of data unit before updating it. Staleness of 5 would mean that a replica can use a value, even if it was last updated 5 logical time units ago.

The authors have devised a number of algorithms geared towards bounding the consistency metrics described above at a replica.

To bound numerical error at a replica, they use a number of push-based algorithms, aiming to measure (conservatively) at replica X how "numerically aberrant" other replicas are compared to itself. One of the algorithms, *Split-weight AE*, keeps track at X, for every other replica Y, how widely the value of a data unit might change, should all updates known to X be propagated and applied on Y. The algorithm pushes X's updates to Y, every time its updates become divergent enough, relative to what it knows of Y's value, to cross a bound numerical bound. For example, if X has a numerical error bound of 3, and has accepted a tentative set of decrements for the data value amounting to a tentative decrease of 4 at Y and increments amounting to a tentative increase of 5 at Y, it would push its updates to Y, before accepting any more updates of the data value. A similar algorithm is described, which aims to bound the relative numerical error between X and other replicas. Note that the algorithms take under consideration the total number of replicas (which all replicas know in advance). So, X assumes (conservatively) that all other replicas have equally "damaging" updates compared to Y, and, therefore, assumed that only $1/(n-1)$ of the overall numerical error is available to it (where n is the total number of replicas). In the example above, if there are 5 replicas in all, X would push its updates to Y when its difference from Y's value became $3/(5-1)$, i.e., its share of the total tolerated numerical error of 3.

Order error is bounded by performing compulsory anti-entropy sessions with other replicas (similar to those described in Bayou [**?**]), every time the outstanding updates at a replica reach the order error bound.

Staleness is also bounded by performing compulsory anti-entropy sessions, every time the oldest accepted write on a value becomes too old relative to the current real-time clock.

The authors have implemented a prototypical middleware layer called TACT, which imposes the appropriate consistency checks based on the parameters set by the applications it helps implement. Three example applications have been built on it: a simplistic airline reservation system, a bulletin board application, and a coarse-grained QoS request redirector. Trial runs of the different applications have been executed between three sites in the wide area.

The overall performance, compared to a reference system implementing strong consistency, seems to be comparable, when certain fine-tuning modifications are made to the algorithms. More specifically, since three different metrics are being bounded, using three different algorithms, in some cases excessive communication results. By optimizing the methods of bounding the three metrics so that they work together (e.g., by sharing the same communication packets to find out information from a remote replica), the authors were able to minimize the overhead of TACT, compared to the monolithic, strongly consistent system.

Issues Although the paper brings up a very very interesting new twist in the search of consistency understanding, it takes a rather shallow approach to evaluating its applicability. More thorough analysis of the scalability issues involved is direly needed.

Also, the paper gives very little insight as to how exactly the bounds used within the algorithm and the resulting observed bounds relate in practise.

However, overall, this was a very interesting, illuminating paper pointing in an exciting direction.

References

[1] William Adjie-Winoto, Elliot Schwarz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, U.S.A., December 1999. ACM SIGOPS. ¿¿¿8¿¿¿.

[2] J. Alves-Foss. An Overview of SNIF: A Tool for Surveying Network Information Flow. In *Proceedings of the First Symposium on Network and Distributed Systems Security*, pages 94–101, San Diego, CA, U.S.A., September 1995. Internet Society. ¿¿¿11¿¿¿.

[3] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *Proceedings of the Information Security Conference (ISC 2001)*, volume 2200 of *Lecture Notes in Computer Science*, pages 379–393, Malaga, Spain, October 2001. Springer. ¿¿¿12¿¿¿.

[4] Ross Anderson. Why Cryptosystems Fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 215–227. ACM Press, 1993. ¿¿¿13¿¿¿.

[5] Ross Anderson, Bruno Crispo, Jong-Hyeon Lee, Harry Manifavas, Vaclav Matyas Jr., and Fabien Petitcolas, editors. *The Global Internet Trust Register*. MIT Press, 1999 edition, March 1999. ISBN 0-262-51105-3 ¿¿¿104¿¿¿.

[6] Ross J. Anderson. The eternity service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology (PRAGOCRYPT 1996)*, Prague, C.Z., 1996. ¿¿¿15¿¿¿.

[7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[8] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network

file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, CO, U.S.A., December 1995. ACM SIGOPS. ¿¿¿14¿¿¿.

[9] N. Anerousis, R. Gopalakrishnan, C.R. Kalmanek, A.E. Kaplan, W.T. Marshall, P.P. Mishra, P.Z. Onufryk, K.K. Ramakrishnan, and C.J. Sreenan. TOPS: An architecture for telephony over packet networks. *IEEE Journal of Selected Areas in Communications*, 17(1), January 1999. ¿¿¿17¿¿¿.

[10] Oguz Angin, Andrew T. Campbell, Michael E. Kounavis, and Raymond R.-F. Liao. The mobiware toolkit: Programmable support for adaptive mobile networking. *IEEE Personal Communications Magazine*, August 1998.

[11] Guido Appenzeller, Kevin Lai, Petros Maniatis, Mema Roussopoulos, Edward Swierk, Xinhua Zhao, and Mary Baker. The mobile people architecture. Technical Report CSL-TR-99-777, Stanford University, January 1999.

[12] Baruch Awerbuch and Christian Scheideler. Robust Distributed Name Service. In *Proceedings of the Third International Peer To Peer Systems Workshop (IPTPS 2004)*, March 2004. ¿¿¿18¿¿¿.

[13] Adam Back. The eternity service. *Phrack Magazine*, 7(51), September 1997. ¿¿¿20¿¿¿.

[14] Tony Ballardie and Jon Crowcroft. Multicast-Specific Security Threats and Counter-Measures. In *Proceedings of the First Symposium on Network and Distributed Systems Security*, pages 2–16, San Diego, CA, U.S.A., September 1995. Internet Society. ¿¿¿22¿¿¿.

[15] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing. In *"Proceedings of the Conference on Communications Architecture & Protocols (SIGCOMM '93)"*, pages 85–95, Ithaca, NY, U.S.A., September 1993. ACM SIGCOMM. ¿¿¿21¿¿¿.

[16] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the Eigth ACM Symposium on Operating Systems Principles*, pages 22–29, Pacific Grove, CA, U.S.A., December 1981. ACM SIGOPS. ¿¿¿23¿¿¿.

[17] Forest Baskett, John H. Howard, and John T. Montague. Task communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, West Lafayette, IN, U.S.A., November 1977. ACM SIGOPS.

[18] Steven M. Bellovin and Michael Merrit. Limitations of the Kerberos Authentication System. *Computer Communications Review*, 20(5):119–132, October 1990. ¿¿¿23¿¿¿.

[19] Josh Benaloh and Michael de Mare. Efficient broadcast time-stamping. Technical Report TR-MCS-91-1, Clarkson University, Department of Mathematics and Computer Science, April 1991. ¿¿¿26¿¿¿.

[20] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972. ¿¿¿28¿¿¿.

[21] Mark Berg, Marc van Kreveld, Mark de Overmars, and Otfried Schwarzkopf. *Computational Geometry by Example*. Unpublished, draft 0.1 edition, 1996.

[22] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–273, Pittsburgh, PA, USA, May 1993. IEEE, IEEE Computer Society Press. ¿¿¿31¿¿¿.

[23] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. ¿¿¿29¿¿¿.

[24] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chamgers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, U.S.A., December 1995. ACM SIGOPS. ¿¿¿32¿¿¿.

[25] Ken Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999. ¿¿¿34¿¿¿.

[26] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC, U.S.A., December 1993. ACM SIGOPS.

[27] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982. ¿¿¿36¿¿¿.

[28] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. ¿¿¿39¿¿¿.

[29] Matt Blaze. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16. ACM Press, 1993. ¿¿¿40¿¿¿.

[30] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE, May 1996. ¿¿¿42¿¿¿.

[31] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970. ¿¿¿43¿¿¿.

[32] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 85–97, Dallas, TX, U.S.A., October 1998. ¿¿¿45¿¿¿.

[33] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 143–156, Saint-Malo, France, December 1997. ACM SIGOPS. ¿¿¿48¿¿¿.

[34] Ahto Buldas, Peeter Laud, Helger Lipmaa, and Jan Villemson. Timestamping with Binary Linking Schemes. In Hugo Krawczyk, editor, *Advances on Cryptology (CRYPTO 1998)*, volume 1462 of *Lecture Notes*

in Computer Science, pages 486–501, Santa Barbara, USA, August 1998. Springer. ¿¿¿51¿¿¿.

[35] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. Optimally Efficient Accountable Time-Stamping. In Yuliang Zheng and Hideki Imai, editors, *Proceedings of Public Key Cryptography 2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 293–305, Melbourne, Australia, January 2000. Springer. ¿¿¿51¿¿¿.

[36] Michael Burrows and Martín Abadi. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990. ¿¿¿52¿¿¿.

[37] Ramón Cáceres and Liviu Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[38] Jakob Carlström and Raphael Rom. Application-aware Admission Control and Scheduling in Web Servers. In *IEEE INFOCOM*, pages 506–515, New York, NY, USA, June 2002. ¿¿¿54¿¿¿.

[39] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *Proceedings of the Fifth Usenix Symposium on Operating Systems Design and Implementation*, pages 299–314, Boston, MA, U.S.A., December 2002. ¿¿¿58¿¿¿.

[40] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, U.S.A., February 1999. USENIX Association. ¿¿¿55¿¿¿.

[41] K. Mani Chandy and Eve M. Schooler. Designing directories in distributed systems: A systematic framework. In *Proceedings of the Workshop on Multimedia and Collaborative Environments, High Performance Distributed Computing Conference*, Syracuse, NY, U.S.A., August 1996. ¿¿¿64¿¿¿.

[42] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Faults containment for shared-

memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, Copper Mountain Resort, CO, U.S.A., December 1995. ACM SIGOPS. ¿¿¿66¿¿¿.

[43] David Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981. ¿¿¿68¿¿¿.

[44] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 74–83, Cambridge, MA, U.S.A., October 1996. ACM SIGPLAN. ¿¿¿70¿¿¿.

[45] Yan Chen, Adam Bargteil, David Bindel, Randy H. Katz, and John Kubiatowicz. Quantifying Network Denial of Service: A Location Service Case Study. In Sihan Qing, Tatsuaki Okamoto, and Jianying Zhou, editors, *Proceedings of the Third International Conference on Information and Communications Security*, volume 2229 of *Lecture Notes in Computer Science*, pages 340–351, Xian, China, November 2001. Springer. ¿¿¿71¿¿¿.

[46] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, Monterey, CA, U.S.A., November 1994. USENIX Association.

[47] David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989. ¿¿¿72¿¿¿.

[48] Ludmila Cherkasova and Peter Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685, June 2002. ¿¿¿76¿¿¿.

[49] Gihwan Cho. A location management scheme supporting route optimization for mobile hosts. *Journal of Network and Systems Management*, 6(1), March 1998.

[50] Gihwan Cho and Lindsay F. Marshall. An efficient location and routing scheme for mobile computing environments. *IEEE Journal on Selected Areas in Communications*, 13(5), June 1995.

[51] E. G. Jr. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971. ¿¿¿77¿¿¿.

[52] Edith Cohen and Haim Kaplan. Proactive caching of DNS records: Approaches and algorithms, 1999. ¿¿¿80¿¿¿. Submitted. On-line: http://www.research.att.com/ edith/Papers/dns.ps.

[53] Microsoft Corporation. Enterprise identity management within the digital nervous system. On-line: http://www.microsoft.com/windows/server/Eval/strategic/EIM.asp, July 1999. ¿¿¿**??**¿¿¿.

[54] R. C. Daley and J. B. Dennis. Virtual memory, processes and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[55] Peter B. Danzig, Katia Obraczka, and Anant Kumar. An analysis of wide-area name server traffic: A study of the internet domain name system. In *"Proceedings of the Conference on Communications Architecture & Protocols (SIGCOMM '92)"*, pages 281–292, Baltimore, MD, U.S.A., August 1992. ACM SIGCOMM. ¿¿¿81¿¿¿.

[56] Don Davis. Compliance Defects in Public-Key Cryptography. In *Proceedings of the 6th USENIX Security Symposium*, pages 171–178, San Jose, CA, USA, July 1996. USENIX Association. ¿¿¿82¿¿¿.

[57] Drew Dean and Adam Stubblefield. Using Client Puzzles to Protect TLS. In *Proceedings of the 10th USENIX Security Symposium*, pages 1–8, Washington, D.C., USA, August 2001. USENIX Association. ¿¿¿84¿¿¿.

[58] Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Brent B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, CA, USA, December 1994. ¿¿¿85¿¿¿.

[59] Dorothy E. Denning and Peter J. Denning. Data security. *ACM Computing Surveys*, 11(3):227–249, September 1979. ¿¿¿87¿¿¿.

[60] Peter J. Denning. Working sets, past and present. *IEEE Transactions on Software Engineering*, 6(2):64–84, March 1980. ¿¿¿88¿¿¿.

[61] John R. Douceur. The Sybil Attack. In *Proceedings of the First International Peer To Peer Systems Workshop (IPTPS 2002)*, San Diego, CA, USA, March 2002. ¿¿¿89¿¿¿.

[62] Christian Dreke. Introduction to personal communication on the internet. *Intel Technology Journal*, Q4, August 1999. On-line: `http://developer.intel.com/technology/itj/q31999/articles/art_2.htm` ¿¿¿91¿¿¿.

[63] D. Eastlake. RFC 2535: Domain Name System Security Extensions, March 1999.

[64] Carl M. Ellison. Establishing Identity Without Certification Authorities. In *Proceedings of the 6th USENIX Security Symposium*, pages 67–76, San Jose, CA, USA, July 1996. USENIX Association. ¿¿¿92¿¿¿.

[65] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, U.S.A., December 1995. ACM SIGOPS.

[66] R. J. Feiertag and E. I. Organick. The MULTICS input/output system. In *Proceedings of the Third ACM Symposium on Operating Systems Principles*, pages 35–41, Palo Alto, CA, U.S.A., 1971. ACM SIGOPS. ¿¿¿94¿¿¿.

[67] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery (JACM)*, 32(2):374–382, April 1985. ¿¿¿94¿¿¿.

[68] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating*

Systems Principles, pages 38–51, Saint-Malo, France, December 1997. ACM SIGOPS.

[69] Armando Fox and Eric Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of HotOS-VII*, Rio Rico, AZ, U.S.A., March 1999. IEEE. ¿¿¿98¿¿¿.

[70] Armando Fox, Steve Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 160–170, Cambridge, MA, U.S.A., October 1996. ACM SIGPLAN. ¿¿¿96¿¿¿.

[71] Christopher W. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, pages 242–246, Atlanta, GA, U.S.A., May 1999. ACM SIGPLAN. ¿¿¿99¿¿¿.

[72] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and DOn'ts of Client Authentication on the Web. In *Proceedings of the 10th USENIX Security Symposium*, pages 251–266, Washington, D.C., USA, August 2001. USENIX Association. ¿¿¿100¿¿¿.

[73] James M. Galvin. Public Key Distribution with Secure DNS. In *Proceedings of the 6th USENIX Security Symposium*, pages 161–170, San Jose, CA, USA, July 1996. USENIX Association. ¿¿¿102¿¿¿.

[74] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pages 150–162, Pacific Grove, CA, U.S.A., December 1979. ACM SIGOPS. ¿¿¿103¿¿¿.

[75] Jim Gray. The transaction concept: Virtues and limitations. In *Very Large Data Bases, 7th International Conference*, pages 144–154, Cannes, France, September 1981. IEEE, IEEE Press.

[76] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus

replicated file system. In *USENIX Summer 1990 Technical Conference Proceedings*, pages 63–71, Anaheim, CA, U.S.A., 1990. USENIX. ¿¿¿105¿¿¿.

[77] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991. ¿¿¿106¿¿¿.

[78] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC, U.S.A., December 1993. ACM SIGOPS. ¿¿¿109¿¿¿.

[79] S. Hardcastle-Kille. RFC 1484: Using the OSI directory to achieve user friendly naming (OSI-DS 24 (v12)), July 1993. Obsoleted by RFC1781 [**?**]. ¿¿¿183¿¿¿.

[80] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[81] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, Asheville, NC, U.S.A., December 1993. ACM SIGOPS. ¿¿¿110¿¿¿.

[82] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–199, Boston, MA, U.S.A., October 1992. ACM SIGPLAN. ¿¿¿111¿¿¿.

[83] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988. ¿¿¿112¿¿¿.

[84] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. ¿¿¿113¿¿¿.

[85] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. ¿¿¿114¿¿¿.

[86] Kevin John Soo Hoo. *How much is enough? A risk management approach to computer security*. PhD thesis, Stanford University, Department of Management Science, Stanford, CA, USA, 2000. ¿¿¿203¿¿¿.

[87] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, September 2003. ¿¿¿116¿¿¿.

[88] Jon Inouye and Jim Binkley. Physical media independence: System support for dynamically available network interfaces. Technical Report CSE-97-001, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, January 1997.

[89] Jon Inouye, Jim Binkley, and Jonathan Walpole. Dynamic network reconfiguration support for mobile computers. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 13–22, Budapest, Hungary, September 1997. ACM/IEEE.

[90] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. IP-based protocols for mobile internetworking. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols*, volume 21, pages 235–245, Zurich, Switzerland, September 1991. ACM SIGCOMM.

[91] ITU-T. *Recommendation X.500: Data Networks and Open System Communications - Directory*. International Telecommunication Union, 3rd:08/1997 edition, August 1997. ¿¿¿232¿¿¿.

[92] ITU-T. *Recommendation X.501: Data Networks and Open System Communications - Directory: Models*. International Telecommunication Union, 3rd:08/1997 edition, August 1997. ¿¿¿235¿¿¿.

[93] Jan Jannink, Derek Lam, Jennifer Widom, Donald C. Cox, and Narayanan Shivakumar. Efficient and flexible location management techniques for wireless communication systems. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MOBICOM '96)*, pages 38–49, Rye, NY, USA, November 1996. ACM. ¿¿¿117¿¿¿.

[94] Per Johansson, Tony Larsson, Nicklas Hedman, Bartosz Mielczarek, and Mikael Degermark. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 195–206, Seattle, WA, U.S.A., August 1999. ¿¿¿119¿¿¿.

[95] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, Saint-Malo, France, December 1997. ACM SIGOPS.

[96] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. SOS: Secure Overlay Services. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 61–72, 2002. ¿¿¿121¿¿¿.

[97] James J. Kistler and M. Satyanarayanan. Disconnected operations in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992. ¿¿¿123¿¿¿.

[98] Steven R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer 1986 Technical Conference Proceedings*, pages 238–247, Atlanta, GA, U.S.A., 1986. USENIX. ¿¿¿124¿¿¿.

[99] Paul Kocher. On Certificate Revocation and Validation. In *Financial Cryptography (FC 1998)*, volume 1465 of *Lecture Notes in Computer Science*, pages 172–177. Springer, 1998.

[100] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. A cluster-based approach for routing in dynamic networks. *Computer Communications Review*, 27(2):49–65, April 1997. ¿¿¿125¿¿¿.

[101] Nancy P. Kronenberg, Henry M. Levy, and William D Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986. ¿¿¿127¿¿¿.

[102] Derek Lam, Yingwei Cui, Donald C. Cox, and Jennifer Widom. A Location Management Technique to Support Lifelong Numbering in Personal Communications Services. *"ACM Mobile Computing and Communications Review"*, 2(1):27–35, January 1998. ¿¿¿129¿¿¿.

[103] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982. ¿¿¿130¿¿¿.

[104] Butler W. Lampson. Designing a global name service. In *Proceedings of the Fifth Annual ACM Symposium on Princiles of Distributed Computing*, pages 1–10, Calgary, AL, Canada, August 1986. ACM. ¿¿¿133¿¿¿.

[105] Butler W. Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 165–182, Pacific Grove, CA, U.S.A., October 1991. ACM SIGOPS, ACM Press. ¿¿¿135¿¿¿.

[106] Butler W. Lampson and David D. Redell. Experiences with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980. ¿¿¿131¿¿¿.

[107] Hugh C. Lauer. Observations on the development of an operating system. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 14–16, Pacific Grove, CA, U.S.A., December 1981. ACM SIGOPS.

[108] Hugh C. Lauer and Roger M. Needham. On the duality of operating systems structures. *Operating Systems Review*, 13(2):3–19, April 1979. ¿¿¿139¿¿¿.

[109] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. http://www.cryptosavvy.com/cryptosizes.pdf, November 1999. ¿¿¿140¿¿¿.

[110] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, U.S.A., October 1991. ACM SIGOPS, ACM Press. ¿¿¿144¿¿¿.

[111] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth P. Birman, and Robert L. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 80–92, Kiawah Island, SC, U.S.A., December 1999. ACM SIGOPS. ¿¿¿145¿¿¿.

[112] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 10 November 1995. ¿¿¿148¿¿¿.

[113] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 135–150, San Diego, CA, U.S.A., October 2000. USENIX Association. ¿¿¿149¿¿¿.

[114] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 191–200, Litchfield Park, AZ, U.S.A., December 1989. ACM SIGOPS. ¿¿¿149¿¿¿.

[115] David Mazières. A toolkit for user-level file systems. In *Proceedings of the USENIX Technical Conference (USENIX 2001)*, pages 261–274, Boston, MA, USA, June 2001. USENIX. ¿¿¿150¿¿¿.

[116] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984. ¿¿¿153¿¿¿.

[117] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the spring system. In *Spring COMPCON 94 Digest of Papers*, pages 122–131, San Francisco, CA, U.S.A., February 1994. IEEE.

[118] P. V. Mockapetris. RFC 1034: Domain names — concepts and facilities, November 1987.

[119] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. In *ACM SIGCOMM 1988 Symposium proceedings on Communications architectures and protocols*, pages 123–133, Stanford, CA, U.S.A., August 1988. ACM SIGCOMM. ¿¿¿155¿¿¿.

[120] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 217–231, Kiawah Island, SC, U.S.A., December 1999. ACM SIGOPS. ¿¿¿156¿¿¿.

[121] Abdelaziz Mounji, Baudouin Le Charlier, Naji Habra, and Denis Zampuniéris. Distributed Audit Trail Analysis. In *Proceedings of the First Symposium on Network and Distributed Systems Security*, pages 102–112, San Diego, CA, U.S.A., September 1995. Internet Society. ¿¿¿158¿¿¿.

[122] Lily B. Mummert, Maria Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, CO, U.S.A., December 1995. ACM/SIGOPS.

[123] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM SIGOPS. ¿¿¿158¿¿¿.

[124] Andrew Myles, David B. Johnson, and Charles Perkins. A mobile host protocol supporting route optimization and authentication. *IEEE Journal on Selected Areas in Communications*, 13(5):839–849, June 1995. special issue on "Mobile and Wireless Computing Networks".

[125] Moni Naor and Kobbi Nissim. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, USA, January 1998. USENIX Association. ¿¿¿159¿¿¿.

[126] Roger M. Needham. Denial of Service. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 151–153. ACM Press, 1993. ¿¿¿162¿¿¿.

[127] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978. ¿¿¿160¿¿¿.

[128] Michael N. Nelson and Sanjay R. Radia. A uniform name service for spring's UNIX environment. In *USENIX Winter 1994 Technical Conference Proceedings*, pages 335–348, San Francisco, CA, January 1994. USENIX. ¿¿¿163¿¿¿.

[129] Gerald W. Neufeld. Descriptive names in x.500. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols*, pages 64–71, Austin, TX, U.S.A., September 1989. ACM SIGCOMM. ¿¿¿163¿¿¿.

[130] Peter G. Neumann. What's in a name? *Communications of the ACM*, 35(1):186, January 1992. ¿¿¿165¿¿¿.

[131] James W. O' Toole and David K. Gifford. Names should mean What, not Where. In *5th ACM Workshop on Distributed Systems*. ACM, September 1992. ¿¿¿165¿¿¿.

[132] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus - an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 58–68, Asheville, NC, U.S.A., December 1993. ACM SIGOPS.

[133] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: the common object request broker architecture*. Pretice Hall, Inc., Upper Saddle River, New Jersey 07458, USA, 1st edition, 1996. ¿¿¿166¿¿¿.

[134] C. Partridge, T. Mendez, and W. Milliken. RFC 1546: Host anycasting service, November 1993. Status: INFORMATIONAL. ¿¿¿184¿¿¿.

[135] Oren Patashnik. Bibtexing. An on-line manual on BibTeX, February 1988.

[136] Charles Perkins, Andrew Myles, and David B. Johnson. IMHP: A mobile host protocol for the internet. *Computer Networks And ISDN Systems*, 27(3):479–491, 1994.

[137] Charles E. Perkins, Andrew Myles, and David B. Johnson. The internet mobile host protocol (IMHP). In *INET '94/JENC5*, pages 642–1 – 642–9. Internet Society, June 1994.

[138] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the First ACM Workshop on Hot Topics in Networks*, Princeton, NJ, USA, October 2002. ¿¿¿170¿¿¿.

[139] Larry L. Peterson. The profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988. ¿¿¿166¿¿¿.

[140] Alan Pope. *The CORBA reference guide: understanding the common object request broker architecture*. Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA, 1997.

[141] Sanjay Radia, Peter Madany, and Michael L. Powell. Persistence in the spring system. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 12–23, Asheville, NC, U.S.A., December 1993. IEEE, IEEE Computer Society Press. ¿¿¿171¿¿¿.

[142] Sanjay Radia, Michael N. Nelson, and Michael L. Powell. The Spring Name Service. Technical Report SMLI TR-93-16, Sun Microsystems Laboratories, Inc., M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, November 1993. ¿¿¿176¿¿¿.

[143] Sanjay R. Radia. Naming policies in the spring system. In *Proceedings of IEEE Workshop on Services for Distributed and Networked Environments*, pages 164–171, Prague, Czech Republic, June 1994. IEEE, IEEE Computer Society Press. ¿¿¿178¿¿¿.

[144] Suchitra Raman and Steven McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols*, pages 15–25, Cambridge, MA, U.S.A., September 1999. ACM SIGCOMM. ¿¿¿178¿¿¿.

[145] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, November 1998. ¿¿¿181¿¿¿.

[146] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[147] Ronald L. Rivest. Can we eliminate certificate revocation lists? In *Financial Cryptography*, pages 178–183, 1998. ¿¿¿185¿¿¿.

[148] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 1–15, Pacific Grove, CA, U.S.A., October 1991. ACM SIGOPS, ACM Press. ¿¿¿187¿¿¿.

[149] David S. H. Rosenthal. Evolving the Vnode interface. In *USENIX Summer 1990 Technical Conference Proceedings*, pages 107–117, Anaheim, CA, U.S.A., 1990. USENIX. ¿¿¿188¿¿¿.

[150] Mema Roussopoulos, Petros Maniatis, Kevin Lai, Guido Appenzeller, and Mary Baker. Person-level Routing in the Mobile People Architecture. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 165–176, Boulder, CO, U.S.A., October 1999. USENIX Association.

[151] Aviel D. Rubin. Trusted Distribution of Software Over the Internet. In *Proceedings of the First Symposium on Network and Distributed Systems Security*, pages 47–53, San Diego, CA, U.S.A., September 1995. Internet Society. ¿¿¿189¿¿¿.

[152] Sandberg Russel, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network FIlesystem. In *USENIX Summer 1985 Technical Conference Proceedings*, pages 119–130, Portland, OR, U.S.A., June 1985. USENIX. ¿¿¿189¿¿¿.

[153] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984. ¿¿¿194¿¿¿.

[154] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972. ¿¿¿191¿¿¿.

[155] L. H. Seawright and R. A. MacKinnon. VM/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[156] David G. Simmons and Ronald Wilkins. Network Event Recording Device: An Automated System for Network Anomaly Detection and Notification. In *Proceedings of the First Symposium on Network and Distributed Systems Security*, pages 87–93, San Diego, CA, U.S.A., September 1995. Internet Society. ¿¿¿198¿¿¿.

[157] Emil Sit and Robert Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *Proceedings of the First International Peer To Peer Systems Workshop (IPTPS 2002)*, San Diego, CA, USA, March 2002. ¿¿¿198¿¿¿.

[158] Alan Jay Smith. Input/output optimization and disk architectures: A survey. *Performance and Evaluation*, 1:104–117, 1981. ¿¿¿200¿¿¿.

[159] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 234–240, Budapest, Hungary, September 1997. ACM/IEEE. ¿¿¿208¿¿¿.

[160] Jennifer G. Steiner, Clifford Neumann, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Winter 1988 Technical Conference Proceedings*, pages 191–202. USENIX, February 1988.

[161] Mark Stemm. Vertical handoffs in wireless overlay networks. Master's thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, U.S.A., 1996.

[162] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Professional Computing Series. Addison-Wesley, 1st edition, 1994.

[163] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981. ¿¿¿211¿¿¿.

[164] Michael Stonebraker, Deborah DuBourdieux, and William Edwards. Problems in supporting data base transactions in an operating system transaction manager. *Operating Systems Review*, 19(1):6–14, January 1985. ¿¿¿213¿¿¿.

[165] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 184–200, Copper Mountain Resort, CO, U.S.A., December 1995. ACM SIGOPS. ¿¿¿214¿¿¿.

[166] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997. ¿¿¿217¿¿¿.

[167] Fumio Teraoka, Kim Claffy, and Mario Tokoro. Design, implementation, and evaluation of virtual internet protocol. Technical Report SCSL-TR-92-003, Sony Computer Science Laboratory Inc., 3-14-13 Higashi-gotanda, Shinagawa-ku, Tokyo, 141 Japan, March 1992.

[168] Fumio Teraoka and Mario Tokoro. Host migration transparency in IP networks: The VIP approach. Technical Report SCSL-TR-93-001, Sony Computer Science Laboratory Inc., 3-14-13 Higashi-gotanda, Shinagawa-ku, Tokyo, 141 Japan, January 1993.

[169] Fumio Teraoka, Yasuhiko Yokote, and Mario Tokoro. A network architecture providing host migration transparency. Technical Report SCSL-TR-91-004, Sony Computer Science Laboratory Inc., 3-14-13 Higashi-gotanda, Shinagawa-ku, Tokyo, 141 Japan, May 1991.

[170] Douglas B. Terry. Towards a Quality of Service Model for Replicated Data Access. In *Proceedings Second International Workshop on Services in Distributed and Networked Environments (SDNE)*, pages 118–122, Whistler, BC, Canada, June 1995. ¿¿¿221¿¿¿.

[171] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the International*

Conference on Parallel and Distributed Information Systems (PDIS), pages 140–149, Austin, TX, USA, September 1994. ¿¿¿219¿¿¿.

[172] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, CO, U.S.A., September 1995. ACM SIGOPS. ¿¿¿222¿¿¿.

[173] Marvin M. Theimer, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Brent B. Welch. Dealing with Tentative Data Values in Disconnected Work Groups. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 192–195, Santa Cruz, CA, USA, December 1994. ¿¿¿225¿¿¿.

[174] Amin Vahdat, Michael Dahlin, Anderson Thomas, and Amit Aggarwal. Active names: Flexible location and transport of wide-area resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 151–164, Boulder, CO, U.S.A., October 1999. USENIX Association. ¿¿¿226¿¿¿.

[175] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), February 1997.

[176] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70, Bretton Woods, NH, U.S.A., October 1983. ACM SIGOPS. ¿¿¿228¿¿¿.

[177] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 230–243, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM SIGOPS. ¿¿¿230¿¿¿.

[178] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Washington, D.C., USA, August 1999. USENIX Association. ¿¿¿231¿¿¿.

[179] David C. M. Wood, Sean S. Coleman, and Michael F. Schwartz. Fremont: A System for Discovering Network Characteristics and Problems. In *USENIX Winter 1993 Technical Conference Proceedings*, pages 335–348. USENIX, January 1993. ¿¿¿232¿¿¿.

[180] Dong Xuan, Sriram Chellappan, Xun Wang, and Shengquan Wang. Analyzing the Secure Overlay Services Architecture under Intelligent DDoS Attacks. In *Proceedings of the International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004. IEEE. ¿¿¿241¿¿¿.

[181] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 305–318, San Diego, CA, U.S.A., October 2000. USENIX Association. ¿¿¿242¿¿¿.