Selective Notification:
Combining Forms of Decoupled Addressing for Internet-Scale Command and Alert Dissemination

Jonathan C. Hill, John C. Knight[1]

Department of Computer Science,
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740

{jch8f, knight}@cs.virginia.edu

Abstract

By using an information survivability control system, the survivability of critical networked information systems can be enhanced using a variety of fault-tolerance mechanisms. Essential to the effective implementation of such mechanisms is communication from the error detection component to the various application nodes in the network. In this paper, we introduce a technique called Selective Notification for the communication of commands and alerts in very large distributed systems. The technique combines intentional addressing, content addressing and sender qualification in a single decoupled event-delivery mechanism. We show that effective targeted command and alert dissemination is achievable, and that Selective Notification allows systems to apply a wide range of event connectivity policies. We present details of an implementation of Selective Notification and the results of performance assessment experiments. Based on our preliminary performance data, we conclude that Selective Notification can be used to support survivability architectures in Internet-sized systems.

Keywords: Survivable distributed systems, survivability, fault tolerance, event notification, decoupled communications.

1. Contact author

Selective Notification: Combining Forms of Decoupled Addressing for Internet-Scale Command and Alert Dissemination

1. Introduction

Modern computing infrastructure consists of more than just computers running software. Millions of computers hosting software applications must consistently inter-operate. End-to-end business systems, the World Wide Web, defense information systems, national financial infrastructure, and telecommunications infrastructure are all examples of Internet-scale distributed applications. These systems must be kept running and should not be vulnerable to preventable failures.

Keeping systems "alive and functioning" is the goal of a significant body of research known as survivability. *Survivable systems* provide service under a wider range of operating conditions, including system damage, varying performance, threats and attacks [14]. By this measure of survivability, today's large-scale distributed applications are fragile. They work well in relatively benign, static environments, but are prone to failure when conditions change.

One approach to improving the survivability of large-scale distributed applications is through fault tolerance using a monitor/analyze/respond architecture (also known as an *information survivability control system* [18] or survivability architecture). Changes in a distributed application are monitored by a set of sensors [15] that generate events to signal changes associated with faults. An analysis system examines received events to detect errors and determine actions necessary to effect error recovery. It then responds by issuing commands to the distributed application that result in changes to system state. Survivability is effected by detection of events that might degrade or threaten system service, analysis of the resulting system state, and reaction to permit system service to be continued, although possibly in a different or degraded form [15]. In order to deal with different fault types, a given survivability architecture might include more that one monitor/analyze/respond control loop.

Large-scale systems present many challenges in the design of a survivability architecture. With millions of components, it is infeasible to require that components have total knowledge of system state—the state is far too large. The fault-tolerance mechanism must rely on the minimum system knowledge with which it can perform error detection and from which it can synthesize valid error-recovery responses. In minimizing knowledge of the distributed application, the fault response system is said to be *decoupled* from distributed application state.

A second issue is how response commands can be communicated effectively to the distributed application—there are likely to be many node types and very large numbers of nodes. It is this second issue that is the subject of this paper. We introduce a technique called **Selective Notification** that combines intentional addressing [1], content-addressing [2], and sender qualification in a single, decoupled event-delivery mechanism. It is a means of targeting commands effectively from a decoupled error-recovery mechanism to an application system consisting of very large numbers of complex and dynamic components. The primary contributions of the paper are: (1) introduction of the concept of Selective Notification, (2) demonstration of its feasibility and performance; (3) presentation of an implementation; and (4) illustration of its role in the efficient programming of command and alert dissemination in Internet-scale systems.

In Section 2 we discuss the communications issues that arise in more detail, and in Section 3, we explore the state-of-the-art in decoupled event-messaging. In Section 4 we introduce Selective Notification as a mechanism combining decoupled addressing mechanisms. Its implementation is discussed in Section 5. Finally, we explore its performance and scaling capability in Section 6.

2. Survivability and Communication

As an example of the type of problem for which survivability is important, consider an Internet-scale distributed application containing many thousands of Web servers and servelets. Unfortunately, such a system can be damaged in many ways, including by direct attacks from entities such as a fast moving “flash” worm, i.e., a process that exploits software vulnerabilities in an exposed network service to spread throughout a network. Typically, as a worm spreads, it devotes infected resources to infecting other remote and vulnerable services, quickly dominating the resources of an application system. There is a wide variety of other traumas to which systems are vulnerable including coordinated security attacks, terrorist attacks, wide-area environmental damage, and various forms of software failure, and so on.

Using a survivability architecture [15] as illustrated in Figure 1, we can enhance a system’s survivability against many forms of trauma and attacks such as those from flash worms. The architecture contains sensors, fault-detection and response systems, and actuation potential in the application system. With appropriate sensor capabilities and actuation potential within the distributed application, fault detection and response can enable a system to react in a meaningful way to attacks and other faults. An important element of this system design is a mechanism whereby the commands and alerts generated by a fault-response system can be sent to all relevant components within the application. This is complicated by the typical application system's size and by the fact that *command relevance* is dynamic, i.e., which nodes should respond to a command will depend on the state of the system at the time the command is issued.

Continuing our flash worm example, assume a fault-detection system has detected a worm attack against a particular version of Web-server software. It generates an alert relevant to only Web servers running the vulnerable software. Meanwhile, receivers in charge of Web servers can only understand a subset of all possible alerts emitted by fault-response systems, and so the alerts might only be relevant to a subset of potentially applicable Web servers. Finally, not all fault-response systems on a public network are of the same trustworthiness or security level. Thus an alert from our fault-response system might only be relevant to receivers that are accepting commands from a fault-response system with specific trust and security ratings.

It is unreasonable to expect centralized, up-to-date knowledge of Internet-scale system state. None-the-less, it is clear from this example that command and alert relevance in response to a worm attack might depend on the state of the fault-response system, its alert, and the state of Web servers. While it would be possible to broadcast all commands and alerts, requiring receivers to determine their applicability, a significant body of research has explored an alternative: disseminating message events on the basis of their relevance, a mechanism known as *decoupled event communications*. This mechanism reduces and/or abstracts the processing of event relevance to senders and receivers. More importantly, the technique fully decouples senders and receivers so that each focuses on only relevant commands and alerts. The remainder of this

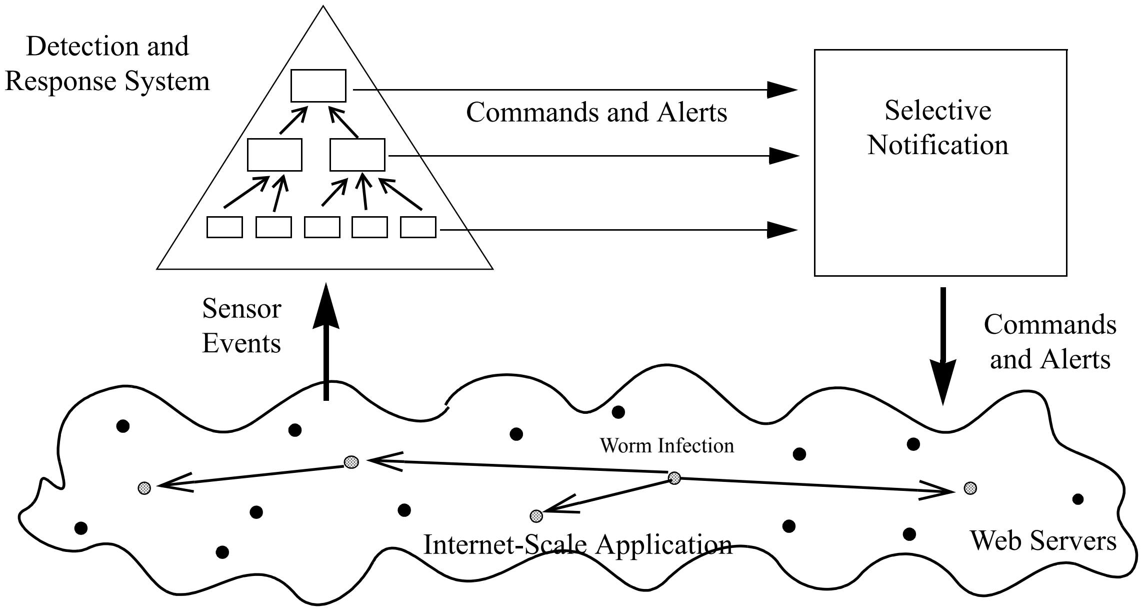


FIGURE 1. Survivability of Internet-scale distributed systems with survivability architecture.

paper is concerned with a means of applying decoupled communications to the full range of targeting requirements of command and alert dissemination in survivability architectures for very large distributed systems.

3. Decoupled Communication of Events

Scalable event dissemination architectures are designed to target event delivery accurately in large distributed systems through event-push architectures [5, 9]. Decoupled event systems are a subclass of event delivery push-systems. Decoupling [6] is the operability of event communications without the requirement that senders or receivers know about each other's quantity, location, or state instantiation. Decoupling can have multiple forms [5], but we focus on spatial decoupling.

In a decoupled system, clients register addresses that are determined by the events that are relevant to them, i.e., the events in which they are interested. The messaging system determines event delivery from these descriptions. We review the most prevalent forms of decoupled event messaging below.

3.1 Publish/Subscribe and Content-Based Addressing

Publish/subscribe is an event-sending paradigm without explicit addresses. Senders inject events into the publish/subscribe system in process is called *publishing*. Receivers are responsible for determining the kinds of events they will receive using descriptions that are called *subscriptions*. The publish/subscribe system delivers all publications to all matching subscriptions. There is no required explicit addressing in publish/subscribe because event content, type, or channel *is* an event's address. When event content is an event's address this is referred to as *content-based addressing* [3]. An example of content-based publish/subscribe is shown in Figure 2(a). Event-content is depicted as shaded triangles and ovals. The figure shows a receiver obtaining an event with content matching a subscription. Note that publish/subscribe [2,4] can be implemented as a distributed algorithm applying *content-based forwarding* [2].

3.2 Intentionally Addressed One-to-Many Messaging

Intentionally addressed, or one-to-many messaging [1], is a form of receiver-descriptive communications. Unlike publish/subscribe, event-messages are sent with explicitly designated address information. However, a message's address is a description of receiver qualities relevant to reception of the message. Receivers are responsible for *advertising* a model of their relevant state. The messaging system forwards messages to matching receivers, as shown in Figure 2(b). Distributed intentional name-space systems can be implemented [1] to provide one-to-many event forwarding by intentional address.

3.3 Sender Qualification

A third means of decoupled communication is sender qualification. This concept is similar to publish/subscribe in that receivers determine the events they will receive. However, receivers describe senders rather than message content. Only events from senders with required state are received. Sender qualification is shown in Figure 2(c), in which a receiver obtains an event from a sender meeting state requirements.

4. Selective Notification

Decoupled addressing mechanisms allow senders and receivers to target events on the basis of unknown but describable remote state, and these mechanisms can be applied for the targeted dissemination of commands and alerts. Returning to our "worm" example system, fault-response systems could target commands to Web servers of a particular software version through intentional addressed one-to-many messaging. Web servers could receive only relevant commands and alerts through content-based publish/subscribe. Finally, Web servers could apply sender qualification to limit reception of commands and alerts to those from trustworthy or authoritative fault response systems.

While each of these mechanisms allows targeting of commands for some aspects of relevance, none supports all of the connectivity requirements above. The command and alert targeting that we require would benefit from simultaneous targeting on the basis of all three decoupled addressing mechanisms. Previous research has shown overlap in the application potential of intentional address [12, 20] and content-

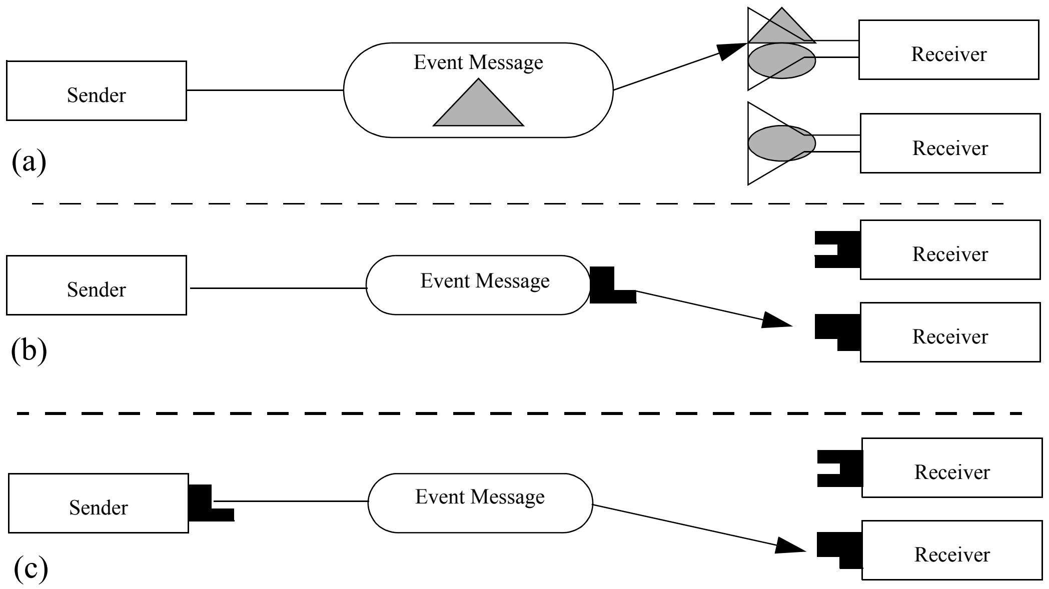


FIGURE 2. Decoupled event communications in (a) content-driven publish/subscribe; (b) intentionally addressed messaging; and (c) sender qualification.

filtering [10, 8]. Selective Notification is an event notification mechanism that combines content-based addressing, intentional addressing, and sender qualification in a unified structure for the delivery of any given event. It has three primary contributions:

- It combines sender, receiver, and message-based, late-bound event forwarding in a single decoupled addressing mechanism.
- It allows all three mechanisms to be applied simultaneously to form directly expressible event-relevancy policies.
- It can be specified in an easy to program descriptive language.

Selective Notification can be implemented in a variety of ways. Our implementation is based on Siena [2], a content-addressed, distributed and scalable publish/subscribe infrastructure, to which we have made a number of extensions and modifications. The resulting communications mechanism supports targeted event dissemination for a more general class of decoupled communication binding policies. As a result, a larger domain of application event dissemination needs are supported by the infrastructure. The performance costs and efficiency costs of their combination are explored in Section 6.

4.1 Combined Addressing Mechanism

The basic forwarding/binding mechanism of Selective Notification is illustrated in Figure 3. In the figure, an array of message sending clients are shown on the left and potential message receiving clients on the right. Both senders and receivers *advertise* their local state. The content of advertised state might include identification of the specific client, classification of the client's capabilities, or details of the client's operating conditions. In the figure, each client's advertised state is represented by an attached solid-black "puzzle-piece".

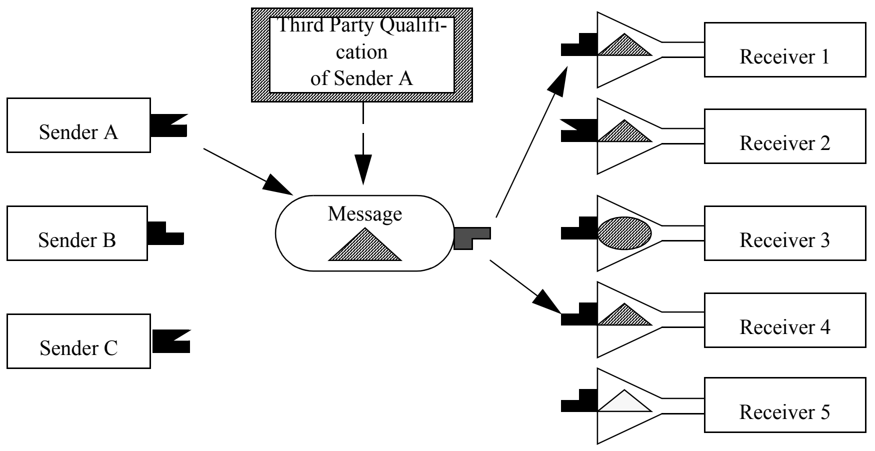


FIGURE 3. The general decoupled communication paradigm of Selective Notification.

As well as communicating clients, authorized third parties can append state to sender qualifications. An authorized third party at the top of the figure appends state to Sender A. The qualification is indicated by the shading on the border of the "Third-Party".

Receivers subscribe to messages with content of interest. The characteristics that define content of interest is depicted by the shape contained in the "funnel" attached to potential receivers. Receivers may also require sender qualifications. These are presented in the figure by the shading of the shape in the funnel.

Senders issue events to the Selective Notification system—a sender can construct arbitrary event content. Figure 3 shows Sender A emitting an event with "triangle" content. Only receivers subscribed to triangle content will receive the event message. Sender A has been qualified, by a third party, with "dark shading." Receivers requiring diagonally-shaded senders can receive this event. Finally, Sender A has addressed the event to only receivers displaying "elbow-shaped" state. Based on the state in the system, the event-message will only be delivered to receivers 1 and 4. Receiver 2 does not have the client state required by the event, receiver 3 does not subscribe to the event's content, and receiver 5 requires sender qualifications not met by Sender A.

4.2 An Overview of Selective Notification Constructs

Selective Notification depends on client and event state, and descriptions of the relevance of this state. The data constructs synthesizing these concepts are shown in Figure 4. State and its relevance are represented with attributes that are named and typed. The attribute system employed is that of the Siena publish/subscribe system [2]. Its contributed elements are shown as thickly bordered boxes.

Clients each possess a model of their potential states that consists of a list of typed, named attributes. Clients advertise their current state by presenting their model and values for their attributes. Events are also

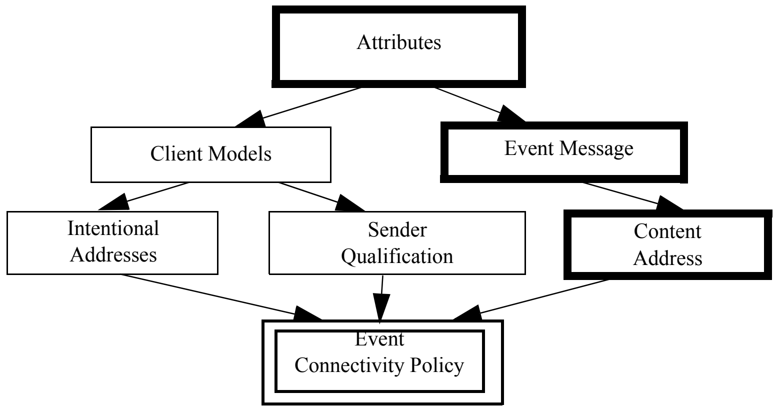


FIGURE 4. The composition of Selective Notification and Siena elements.

constructed from attributes, and an event is a list of typed, named, attributes with an assigned value for each attribute. A sender can issue an event by providing a list of typed, named attributes and their values.

Constraints on attributes represent event relevancy. Thus, intentional address, sender qualification, and content address are all represented as Boolean expressions over constraints on the values of attributes. Evaluation of a Boolean expression to *true* for a set of attribute values indicates that an event is relevant. An intentional address is a Boolean expression evaluated over receiver-model attributes. Likewise, sender qualifications are Boolean expressions evaluated over sender-model attributes. Content addresses are Boolean expressions over event-content attributes. Receivers register sender qualifications and event addresses in order to receive events from qualified senders. Senders attach optional intentional addresses to events in order to send events to a subset of potential receivers. Together, these constraints define connectivity policies.

4.3 Command and Alert: Surviving a Fast Moving Internet Worm

Returning to our worm example once again, we assume that an error-detection system that is part of the survivability architecture will detect worm attacks through the collection of sensor events. It will use Selective Notification to effect the necessary command and alert mechanism so that it can target the affected system without having to maintain detailed state knowledge for all components in the application. In the rest of this section, we present an example specification for the command and alert dissemination mechanism. Essentially, this is the “program” used by the Selective Notification mechanism.

Modelling System State

Our model of Web servers, i.e., the state that Web servers advertise, is:

```
<Model WebService>=     {String  application;                  String  application_version;
                        String  serviceIPAddress;              int  servicePort;
                        DomainedSet{docs, cgi, xml}  services; float  load}
```

Recall that models are named elements consisting of typed, named attributes. In our implementation, supported attribute types are Boolean, integer, floating point, string, and finite-domained set. Every Web service in our example application presents an instance of this model to Selective Notification. An instance of a model is an assignment of a value to each model attribute, so a Web server somewhere might supply:

```
<WebService>      =     {application=”IIS”;                    application_version=2.4.0;
                        serviceIPAddress=”128.142.55.55”;      servicePort=8080;
                        services={docs, cgi};                  load=0.39}
```

This Web server is free to change its attribute values at any time, and this is one of the major sources of dynamic state that was mentioned above. For example, this Web server will periodically update its load attribute with its latest load calculation. System controllers and managers might be presenting clients on behalf of Web-service applications.

Now suppose that our application shown in Figure 1 is controlled by more than one control loop in its survivability architecture. There might be, for example, a collection of regional fault-response systems and, separately, a national system. These systems are not allowed to define their own region of command or trustworthiness. Instead, this state is assigned to fault-response systems by authorized third parties such as trust managers and command authorizers. For example, a Northwest regional controller might be assigned the following sender qualifications by authorized third parties:

```
<Model FRSystem> =        {String _command_region="northwest";        int _trust_level=4}
```

Restricted sender qualification allows a tiered-authority model of sender state enforcement, so that increasingly critical state can be managed by increasingly trustworthy elements. We note that the integrity of the Selective Notification service is an important issue but is not discussed here.

Connectivity Policies

Receiver connectivity policies that command and alert Web Services are Boolean expressions conjoining content filters and sender qualifications. A policy may consists of a series of such conjunctions in disjunction. For example, the Web Services in our hypothetical application might register the following connectivity policy:

```
[<Event> :    {alert==ANY
              AND threat_level>==4}
              AND <Sender=="ControlSystem">:{_command_region=="northwest" AND _trust_level>=2}]
OR
[<Event> :    {command==ANY}
              AND <Sender=="ControlSystem">:{_command_region=="national"} ]
```

which translates to "Observes command events from any national control system and alerts greater than or equal to threat-level 4 from Northwest controllers with trust rating greater than or equal to 2". Once in place, received events from senders are those for which evaluation of the connectivity policy expression is *true*. Thus, Web servers only receive understandable commands and alerts from qualified commanders.

Command and Alert Events

Assume that a worm has begun propagating through our distributed application. A fault-response system in the Northwest is the first to detect the worm infection. It determines that the sensor events are all coming from Web servers running version 2.4 of "IIS." First, it reports this to a national fault-response system. Then, it issues a worm alert as an event:

```
Event:{alert="worm";  threat_level=4; target="IIS"; version="2.4.*"}
```

Since events are collections of attributes, any receiver of this event will be able to obtain the information associated with the event by examining its attributes.

Given that Web servers are enforcing the policy defined in the previous subsection, all Web servers in the Northwest region will receive this alert. From the alert they can determine if they might be vulnerable to attack. Meanwhile, the worm continues to infect the distributed application. Our national fault-response system attempts to halt the attack with commands to Web servers. The national fault-response system has determined that the worm is spreading through a vulnerability exposed in CGI-scripts running on version 2.4 of IIS Web servers. Therefore, it issues the following command event:

```
<Model==WebServer>:{application==“IIS”  AND  application_version<“2.4”  AND  services include{cgi}}
Event:{command=”disable_cgi”}
```

This event contains a preamble that is an intentional address which selects Web servers that are running “IIS” version 2.4 and support CGI scripts. The event itself is a command for those Web servers to disable CGI elements. The goal of this command is to limit the infection by disabling the worm’s attack vector.

In an attempt to limit the worm’s aggression, the national fault-response system emits another command. It has determined that IIS version 2.4 servers showing sustained load are potentially infected. These servers are ordered to shut-down with the following intentionally addressed command event:

```
<Model==WebServer>:{application=“IIS”  AND  application_version<“2.4”  AND  load>0.9}
Event: {command=”shutdown_now”}
```

In this example, we have seen an alert event and two command events delivered to application components in an Internet-scale system. The connectivity policies at receivers and the intentional addresses at senders define total connectivity policies targeting the commands and alerts. In terms of semantics, all of the relevancy elements in Section 2 have been addressed.

The demonstrated capabilities of decoupled command and alert targeting are:

- *Formal specification*: Boolean expressions over sender, receiver, and event state define connectivity policy as invariants for event reception over system state.
- *Run-time policies*: policies can be invoked and revoked by clients while running.
- *Dynamics of late-bound message forwarding*: The values of attributes describing system state are a run-time property of event clients and third parties, and thus so is event delivery [8, 1].

5. Implementation

Although the basic semantics of Selective Notification provide comprehensive functionality able to support the recovery mechanisms needed for network survivability, such as dealing with a worm infection, the technique will only be useful if the semantics can be implemented in an effective and efficient way.

In order to evaluate Selective Notification, we have developed an implementation in Java as functionality atop, and as extensions within, an implementation of the Siena publish/subscribe system [7]. In this section, we discuss details of the implementation. In the next section, we present the results of experimentation and detailed performance measurements.

Selective Notification operates as a Siena overlay network: a distributed tree of dispatcher applications accepting clients and events. Events are called *notifications* and consist of collections of attributes. Filters over attributes determine the propagation of notifications within the Siena dispatcher overlay network. The result is delivery of all and only relevant events to receivers based on their subscriptions. We obtain intentional address and sender qualification via mapping to content filters, with several required changes to algorithms and extensions within the Siena dispatchers.

5.1 Mapping Selective Notification to Publish/Subscribe

Part of the implementation of Selective Notification is in mapping client models and constraints to the attribute and constraint model of publish/subscribe. Figure 5 shows the mapping of data constructs from Selective Notification (part a) to publish/subscribe (part b). Data elements are represented by shapes. Oper-

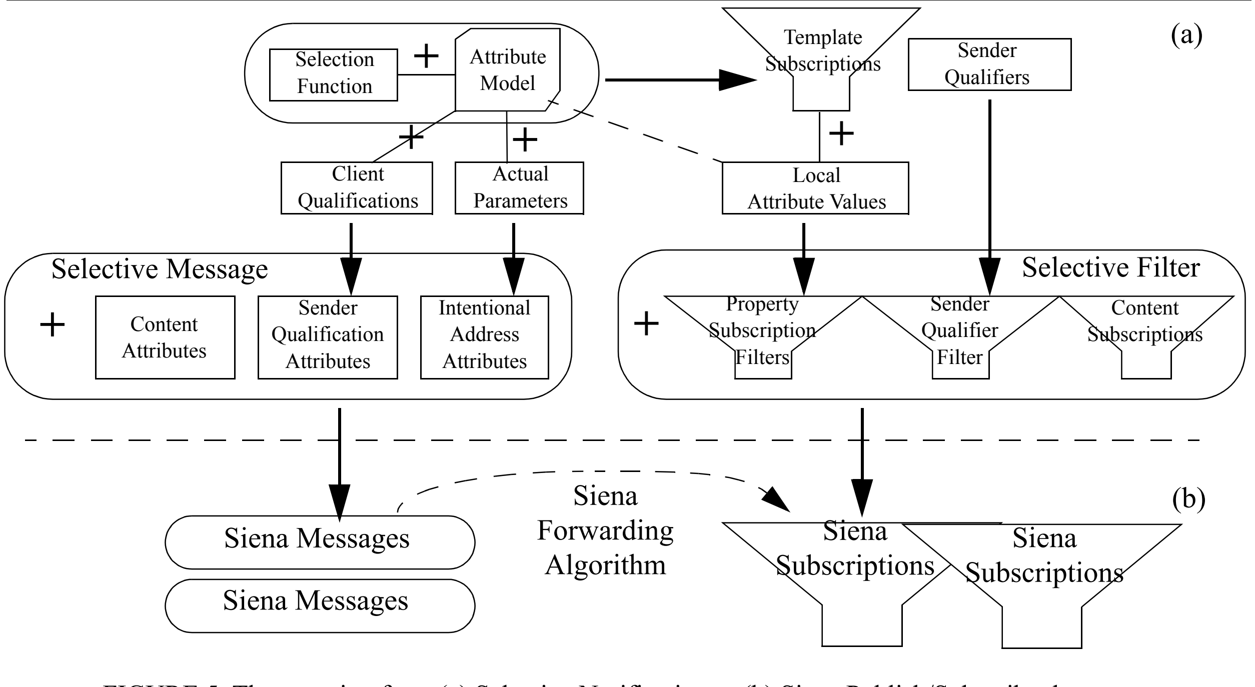


FIGURE 5. The mapping from (a) Selective Notification to (b) Siena Publish/Subscribe data structures.

ations between data elements are represented by "plus-sign" symbols. The product of an operation is connected to its operands by an arrow symbol. In cases of three or more input objects, the inputs are encircled.

Translating from intentional address to receiver subscriptions and event content is the key mapping of Selective Notification. To summarize the operation, clients declare selection functions. Selection functions are parameterized constraints over client models forming a Boolean expression. Senders issue intentional address by providing actual parameter values to these functions. The primary restrictions of selection functions are: (1) required registration at receivers and senders; and (2) the form of their Boolean expression. Boolean expression are limited to comparison terms in which the left-hand operand of a comparison operation must be a formal parameter of the selection function, and the right hand-side must be a client attribute. The previous section applied closed range selection functions generated automatically for any given client model. A more detailed analysis of the mapping from content filters to intentional addressing functions, and their expressability and cost is available in a separate paper [11].

5.2 Modifications to Publish/Subscribe Infrastructure

The second part of the implementation of Selective Notification is based on Siena. Siena, by design, is optimized for scalable and efficient delivery of events through content addressing and so it is not immediately suitable as a basis for the implementation of Selective Notification. In fact, Selective Notification is a pathological application of publish/subscribe because it has the following three properties: (1) all client subscriptions representing attribute models may change rapidly and consistently; (2) client model state of individual client subscriptions represent points in a high-dimensional space, rather than regions of content forming sub-nets; and (3) authorized third parties must be able to establish client state with integrity. We accommodate these conditions with several extensions and modifications to Siena, each of which is explored in the remainder of this section.

Event Persistence

Event persistence gives a lifetime to Siena notifications. Notifications continue to exist in the Siena dispatcher system for the duration of their lifetime. During this time, they follow any new dispatcher paths to clients with newly matching subscriptions. This allows notifications to reach all relevant subscribers even as client state changes.

Persistence is implemented by storing notifications at the dispatcher for their lifetime, and comparing them against new subscriptions. New forwarding paths are discovered by this comparison on a subscription-basis. Its cost is proportional to the time to check a filter matching a subscription, times the rate of subscription changes and the size of the persistent notification store. Book-keeping cost is also incurred.

Filter Coagulation

Siena scales to Internet-sized overlay networks by organizing content filters into a partial order [2]. Thus Siena scales with respect to the extent of partial ordering in the cover relationship between subscriptions in the network. The client state subscriptions of Selective Notification—those used to support intentional addressing—do not cover one another. To achieve scale, it is necessary to combine client state.

Filter coagulation is an algorithm that enforces scalable content sub-net inclusion over client state. In this algorithm, filters for client models at a Siena dispatcher are "wrapped" with a minimal filter that contains all filter instances of that model. Only this minimal, inclusive filter is shared with neighboring dispatchers. The result is an enforced number of filters sent between dispatchers. The cost is a reduction in the precision of intentionally addressed event delivery when compared with content addressing. All events are still delivered to appropriate receivers, but there is significant opportunity for events to follow paths in the dispatcher hierarchy to which the message is not forwarded to any clients thereby reducing the message-efficiency potential of the system. The cost of filter coagulation depends on the rate at which it is performed multiplied by the number of client filters of a given model presented to a dispatcher.

Attribute Authorization and Capability

Clients of Selective Notification must have the authority to define event attributes. This is implemented by inclusion of an authorization and capability mechanism within Siena. This mechanism requires that clients establish sessions with a dispatcher using an account name and password. An account maintains a list of attribute capabilities. This mechanism restricts legal events and filters that can be generated by sessions of the account, and relies on the integrity of the entire dispatcher network.

Third Party Qualifiers

We enable third parties to contribute state to client events. Third parties must be given permission by a client they are to augment. A client requests a permission key from the dispatcher to which it is attached and it can then share this key with trusted third parties. The key grants parties the right to append state to any events generated by the client. The state that a party may append to the client is restricted by the parties capabilities in its session, as discussed above. A client may revoke keys it has requested at any time, although this results in immediate loss of any attribute contributions from third parties utilizing the key.

Channeling

Rather than computing the forwarding path for all events, we allow for some events to follow the paths of previous events. Channel-writing events store their forwarding paths in dispatchers. Channel-reading events apply the forwarding information of past writing events. The latter can reuse the forwarding paths of previous events, as well as apply reception of a previous event as a condition of their delivery to receivers. The latter application is an extension of intentional address with a temporal relationship between events. The computational cost of channeling is proportional to the cost computing the intersection between two sets of forwarding paths. Its storage cost is proportional to the number of channel-writing messages. Writing and reading authority can be restricted by session account.

6. Performance of Selective Notification for Large Distributed Applications

The essential practical challenge with Selective Notification is scale—it is important to determine the overall performance of the technique and how that performance relates to the scale of the target system. The issue of performance is complex, however, because performance metrics need to be defined and measured along the spectrum of dimensions that will affect performance in real systems. From the perspective of general utility, we consider the following to be the critical metrics:

- *Sustainable event delivery time*: the time from event issue to event delivery to all relevant clients.
- *Sustainable event throughput*: the sustainable rate at which events can be issued into the service without overloading the service.

With those metrics in mind, the key dimensions that affect performance are:

- The size of the application system as measured by the total number of nodes operating independently.
- The connectivity policies that describe senders, receivers, and content simultaneously.
- The rate at which the state presented to the Selective Notification mechanism is changing.

In this section, we present the results of experiments to determine these parameters, and we build implementation-performance driven models of scale.

6.1 Experimental Method

Recall that Selective Notification operates as a distributed service and that this service is composed of components called dispatchers connected to form a tree-like overlay network. The performance of Selective Notification depends, therefore, on the performance of this overlay network.

For purposes of experimentation, we used a dedicated network of 128 physical computers each of which was a dual 400 MHz CPU i86 machine running Redhat Linux 6.4 with kernel version 2.2. All software was implemented in Java for runtime 1.4.1x. Communication was performed with TCP sockets over a 100 MBit/sec fully switched Ethernet. For some experiments, we enable and disable features of operation to determine relative costs of algorithm elements. Some of these computers were dedicated to the execution of Selective Notification dispatchers and the remainder were used to execute a hypothetical distributed application. Several application nodes were located on each physical machine and this permitted a total of several thousand client nodes to be constructed. The actual number of clients varied with the details of the experiments being conducted but the most typical number was 3,400.

Clearly, this target system is not of Internet scale and so we cannot test Selective Notification directly over applications with hundreds of thousands or millions of clients. We can, however, build implementation-driven models of performance. Each Selective Notification dispatcher acts through its local clients, parent, and child dispatchers. Therefore, the performance of a dispatcher tree depends on the performance capabilities of a single dispatcher and that of the dispatchers and clients to which it is attached. Given that all dispatchers operate with the same performance capabilities, a model of the steady-state performance capabilities of Selective Notification can be constructed by modeling a single dispatcher. This model will operate under the assumption of optimal network-level communications. This requires that we (1) determine the performance capabilities of a Selective Notification dispatcher, and (2) apply the results to a model of performance for very-large systems.

6.2 Measurement of Dispatcher Performance

We obtain dispatcher performance capabilities by operating and measuring a dispatcher in a controlled environment in which all the input factors affecting performance are controlled. These factors are:

- *Available Resources*: This includes computing hardware, network resource, and the run-time system environment.
- *Event Forwarding Set Size*: The size of the set of sub-dispatchers and clients to which events are forwarded.
- *Event Persistence Lifetime:* The persistence lifetime of event messages.
- *Subscription Change Rate:* The rate at which clients modify their state and the rate at which filter coagulation is performed.
- *Connection Policy Complexity:* The number of attribute constraints registered by clients.

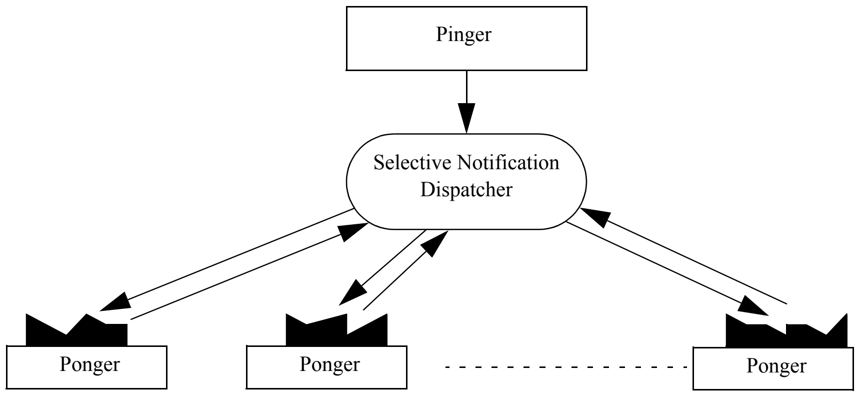


FIGURE 6. An experimental test-bed for performance measurement of Selective Notification.

Our experimental apparatus, illustrated in Figure 6, is a “ping-pong” throughput experiment. A single “Pinger” application sends “ping” messages to “Ponger” applications in sequence. “Ponger” applications respond to the “pinger” with a “pong” message. Additionally, “Ponger” applications generate broadcast-like background messages sent to all other “Pongers.” During the entire experiment, “Ponger” applications randomly change the values of their client models at a specified rate, representing diverse dynamic state change throughout the distributed system. These last two conditions are indicative of a worst-case system behavior with consistently changing client state and the pervasion of “broadcast” commands and alerts. By varying the number of “Ponger” elements, the rate at which they “chatter”, the event-persistence lifetime of “chatter”, the size of the “Ponger” client model, and the rate at which “Ponger”s change their client model instances, we observe performance capability with respect to the input parameters of interest. For purposes of experimentation, each application, including “Pongers”, the “Pinger” and the Selective Notification dispatcher, were run on separate computers.

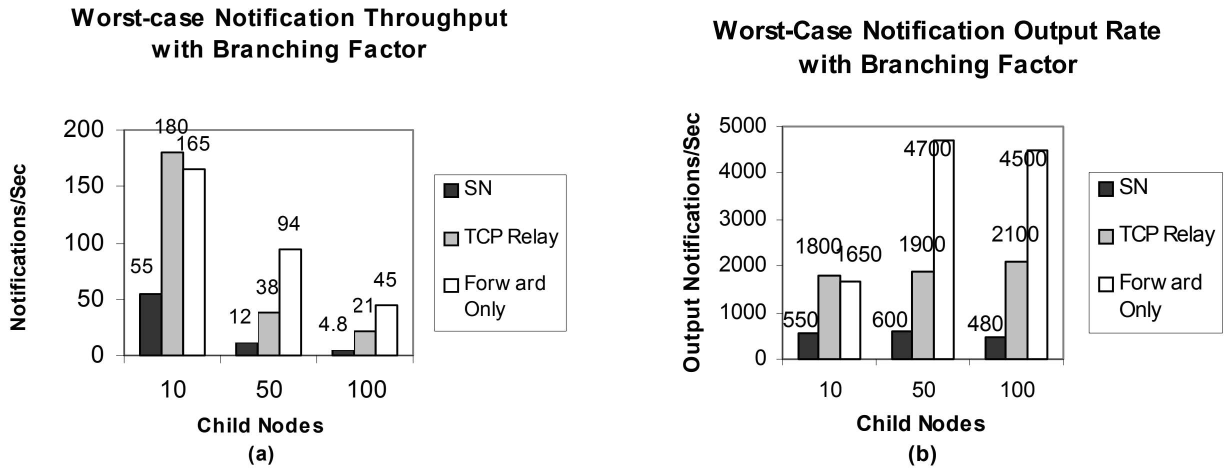


FIGURE 7. (a) Worst-case event throughput with forwarding-set size; (b) Worst case event forwarding rate with forwarding-set size.

Event Throughput and Output Performance

Figure 7(a) shows the maximum throughput rate computed using the “ping-pong” experimental configuration. Figure 7(b) shows the maximum event forwarding rate computed for the same experiments. In each case, the X-axis is the number of “Ponger” clients attached to a Selective Notification dispatcher. Maximum rates were calculated by varying “Ponger” broadcast-chatter rates and determining the point of throughput saturation, i.e., the point where the dispatcher would begin falling behind permanently. This experiment was performed with 10 second client-model attribute updates and coagulation updates, and 60 second persistence lifetimes for notifications. Ping notifications were sent every 2 seconds.

Three versions of the Selective Notification dispatcher were run in separate trials. The first was complete, the second relayed all notifications without forwarding computations, and the third dispatcher performed forwarding computation and simulated infinitely-fast TCP communications to receivers. The results show that the throughput rate of Selective Notification under worst-case forwarding conditions (broadcast) is dominated by the cost of TCP communications, followed by the cost of event forwarding computation. Figure 7(b) demonstrates that these costs are governed by the number of output events that must be produced by the dispatcher. Note that the rate is constant for a given feature set regardless of branching factor. Thus, the primary determination of throughput is the size of the set of potential receivers. The cost is not constant at ten “Pong” nodes for simulated TCP because of Java garbage collection.

Effect of Client Model Size on Performance

The size of client models and the resulting size of targeting notifications has a significant impact on system performance. A client-model’s size is measured by its number of attributes and is proportional to

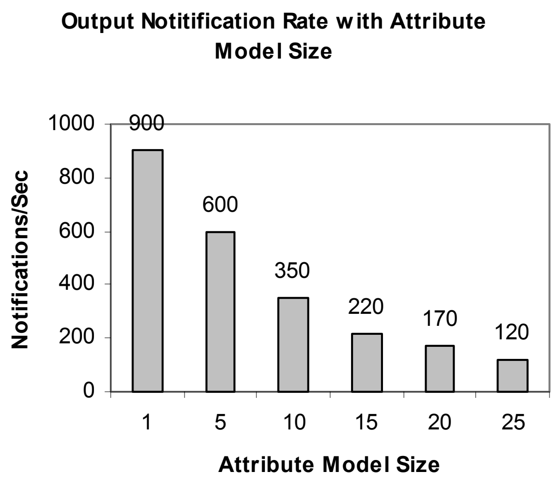


FIGURE 8. The effect of client model size in attributes on the rate of notification forwarding.

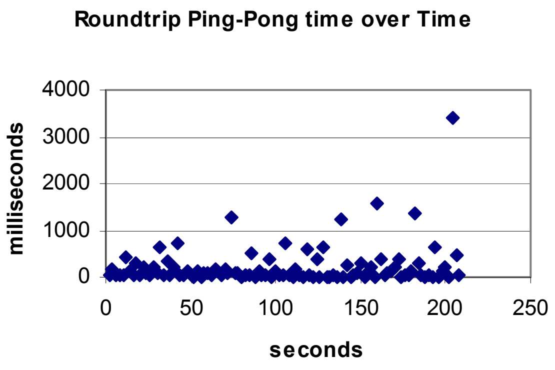
half the size of the default filters generated for intentional addressing. Figure 8(a) shows the output notification rate of Selective Notification with the size of subscription filters. This experiment was performed with 60 second message persistence, 10 second filter coagulation, and 50 “Ponger” applications. The second data point (5 model attributes) corresponds roughly to the size of the attribute models in the experiment from which data was collected in Figure 7. Selective Notification is five times less efficient with 25-attribute client models than with 5-attribute client models.

Round-trip Message Time

Our experiments recorded round-trip message time for “ping” and “pong” messages pairs. Figure 9 shows “ping-pong” time over the course of an experiment in linear and logarithmic scale. The data is from an experiment with 50 “pongers” with attribute changes every ten seconds and notification persistence of 60 seconds. Ten messages were input to the dispatcher per second, so that the system was not processor-saturated. The experiment was run for 200 seconds. Average round-trip time was 220 milliseconds, with a standard deviation of 430 milliseconds. Deviation occurs from persistent notification time-outs, filter coagulation, clusters client model changes, and Java garbage collection. Under the worst case example of these conditions, round-trip time may be as high as 3 or 4 seconds.

6.3 A Model of Scale

From these experiments, it is clear that the throughput of Selective Notification depends heavily on the number of clients and dispatchers connected to a dispatcher, and the client model size. Less important but still significant are the rates of client attribute (model) changes. Using the measurements from the previous section for a dispatcher operating in controlled conditions, we can estimate the maximum throughput potential of Selective Notification in large distributed applications.



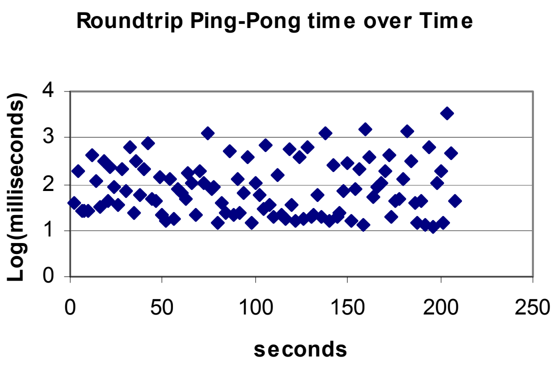


FIGURE 9. Round-trip Ping/Pong-pair message time with Selective Notification.

A hierarchy of Selective Notification dispatchers is needed to reach large numbers clients. As an example, consider a system with a million clients. Figure 10 shows notification worst-case throughput and worst-case delivery time for variations in dispatcher-tree branching factor and dispatcher hardware resource dedication. We consider dedicated dispatcher networks (labelled with “d” in the graph) in which the dispatchers use all computational resources, and peer dispatcher networks (labelled with “p” in the graph.) in which the dispatchers use one tenth of the resources.

Using our current implementation and with a branching factor of a thousand, a dedicated dispatcher tree can support a notification every three seconds and deliver events in four seconds to a million elements. With a branching factor of ten, a peer dispatcher tree can support four notifications per second delivered in 60 seconds. Dedicated dispatchers at the higher levels of the tree and peer dispatchers at lower levels can provide intermediate results for both notification rate and throughput. 10,100 dedicated dispatchers at the base of the tree with a branching factor of 100 followed by peer dispatchers with a branching factor of ten to the clients would result in four notification per second throughput with 24 second delivery time.

These levels of performance suggest immediate utility in the construction of systems designed to cope with a wide variety of attacks. For example, it might be possible to cope with worm attacks in very large networks by sensing changes in resource utilization at the initial stage of an infection and using Selective Notification to shut down vulnerable elements of the network, to partition the network, or to invoke pre-stored protection mechanisms. With the potential speed with which Selective Notification can operate, it would be possible to react aggressively to a worm attack by changing network configurations quickly (within seconds) and then re-establishing network configurations shortly afterwards (within tens of sec-

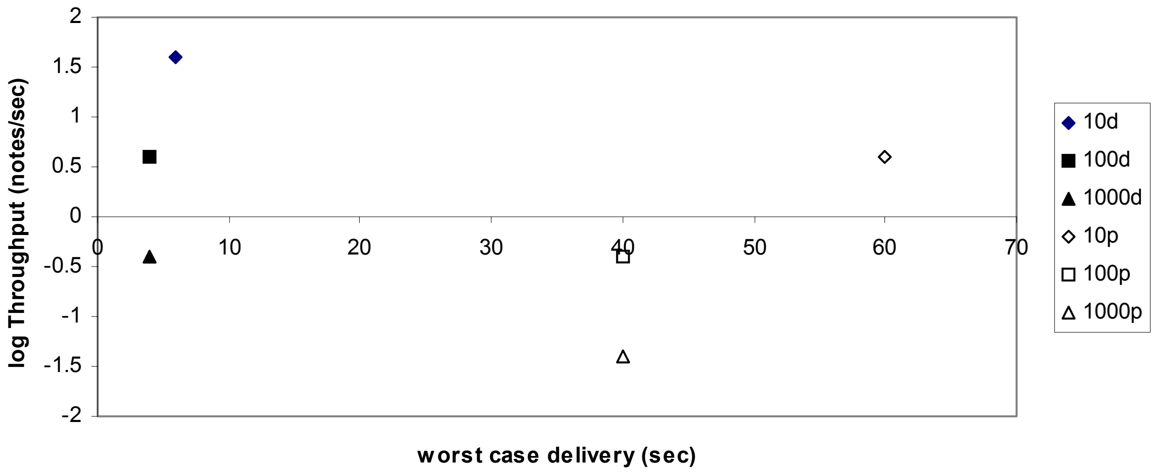


FIGURE 10. Performance design space for varying dispatcher-tree branching factors and for dedicated or peer-hosted dispatchers.

onds) if the circumstances indicate a false alarm or if the infection was halted. In such a case, the users of the network would probably not even be aware of the action.

7. Conclusions

In this paper we have introduced a technique, Selective Notification, that combines intentional addressing, content-addressing and sender qualification in a single decoupled event delivery mechanism, and we have shown that targeted command and alert dissemination is achievable using a combination of these mechanisms. Their combination in Selective Notification allows systems to apply a wider range of event connectivity policies within the forwarding mechanisms of an event-dissemination service.

Applying Selective Notification, fault response mechanisms can better target commands and alerts while reducing the command relevancy workload of senders and receivers. Senders focus on the state required to synthesize meaningful responses, and receivers focus on resolving command conflict issues relevant to their local state knowledge.

Based on the preliminary performance data that we have obtained, we conclude that these policies can be applied to Internet-sized systems that have largely dynamic state. The approach is specialized and intended for the types of communications that we have discussed. The performance will drop to unacceptable levels with various combinations of pathological circumstances such as very large client state models or unbounded rate of client state changes. Such cases are controlled by throttling rates of change and by careful definition of the client state models.

The results are a cost-efficient, flexible means of programming a highly effective fault-tolerance mechanism to enhance the survivability of very large systems. It also provides a reduction in the barrier between top-down, control-driven approaches to survivability, and bottom-up emergence-driven algorithms.

8. Acknowledgments

It is a pleasure to thanks Antonio Carzaniga, Alexander Wolf, and Dennis Heimbigner for their comments on this work and creating the Siena system upon which our implementation is built. This work was supported in part by the Defense Advanced Research Projects Agency under grant N66001-00-8945 (SPAWAR) and the Air Force Research Laboratory under grant F30602-01-1-0503. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force, or the U.S. Government.

9. References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. “The design and implementation of an intentional naming system.” Operating Systems Review, Vol. 34 No. 5, pp 186-2001. Dec. 1999.

[2] A. Carzaniga, D. Rosenblum, A. Wolf. “Achieving scalability and expressiveness in an Internet-scale event notification service.” Symposium on Principles of Distributed Computing, 2000. pp 219-227.

[3] A. Carzaniga, A. Wolf. “Content-based Networking: A New Communication Infrastructure.” NSF Workshop on an Infrastructure for Mobile and Wireless Systems. In conjunction with the International Conference onf Computer Communications and Networks ICCCN. October, 2001.

[4] G. Cugola. E. Di Nitto, A. Fuggetta. “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS.” IEEE Transactions on Software Engineering, Volume: 27 Issue: 9, pp 827 -850. Sept. 2001.

[5] G. Cybenko, B. Brewington “The foundations of information push and pull.” Mathematics of Information. Springer-Verlag, 1998.

[6] P. Eugster, P. Felber, R. Guerraoui, A. Kermarrec. “The Many Faces of Publish/Subscribe.” Microsoft Research Technical Report EPFL, DSC ID:2000104. January 2001.

[7] http://www-serl.cs.colorado.edu

[8] B. Gerkey, M. Mataric. “Murdoch.” Proceedings of the Fourth International Conference on Automated Agents. June 2000.

[9] M. Hauswirth. “Internet-Scale Push Systems for Information Distribution--Architecture, Components, and Communication.” PhD Thesis from Technical University of Vienna, Austria. August 1999.

[10] D. Heimbigner. “Adapting publish/subscribe middleware to achieve Gnutella-like functionality.” Selected Areas in Cryptography, 2001. pp 176-181.

[11] J. Hill. “Site-Selective Notification for Distributed Systems.” Department of Computer Science, University of Virginia Technical Report CS-2002-06.

[12] C. Intanagonwiwat, R. Govindan, D. Estrin. “Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks.” Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM '00). August 2000.

[13] “Jini Architectural Overview.” Sun Microsystems Technical White Paper, 1999.

[14] J. Knight, E. Strunk, K. Sullivan. “Towards a Rigorous Definition of Information System Survivability.” DISCEX 2003, Washington DC, IEEE Press.

[15] J.C. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, M. Gertz. “The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications.”, Intrusion Tolerance Workshop, Dependable Systems and Networks (DSN 2002), Washington DC, June 2002.

[16] A. Rowstron, A. Kermarrec, M. Castro. P. Druschel. “The Design of a Large-Scale Event Notification Infrastructure.” Networked Group Communication, 2001. pp 30-43.

[17] JMS. Java Messaging Service: http://java.sun.com/products/jms/

[18] K. Sullivan, J. Knight, X. Du, S. Geist. “Information Survivability Control Systems.” Twenty-first International Conference on Software Engineering, IEEE Computer Society Press. May 1999.

[19] B. Toone, M. Gertz, P. Devanbu. “Trust Mediation for Distributed Information Systems.” 18th International Information Security Conference. May 2003.

[20] W. Vogels, C. Re, R. Renesse, K. Birman. “A Collaborative Infrastructure for Scalable and Robust News Delivery.” Proceedings of the IEEE Workshop on Resource Sharing in Massively Distributed Systems (RESH'02). July 2002.