Scalable Resource Control in Active Networks

Kostas G. Anagnostakis, Michael W. Hicks, Sotiris Ioannidis,
Angelos D. Keromytis, and Jonathan M. Smith

Distributed Systems Laboratory
Department of Computer and Information Science, University of Pennsylvania
200 S. 33rd Street, Philadelphia, PA 19104-6389, USA
{anagnost,mwh,sotiris,angelos,jms}@dsl.cis.upenn.edu

Abstract. The increased complexity of the service model relative to store-and-forward routers has made resource management one of the paramount concerns in active networking research and engineering. In this paper, we address two major challenges in scaling resource management to many-node active networks. The first is the use of market mechanisms and trading amongst nodes and programs with varying degrees of competition and cooperation to provide a scalable approach to managing active network resources. The second is the use of a trust-management architecture to ensure that the participants in the resource management marketplace have a policy-driven "rule of law" in which marketplace decisions can be made and relied upon. We have used lottery scheduling and the Keynote trust-management system for our implementation, for which we provide some initial performance indications.

1 Introduction

Resource management is a major challenge for active networks because of their increased flexibility. Most processing in current networks is simple packet forwarding, meaning that most packets require the small fixed cost of a routing table lookup and a copy. In the grand vision of active networking, the network provides services that are customizable by its users. This means that packet processing is much more complicated because it is, at least in part, user-specified. Therefore, and in the absence of a more sophisticated resource management model, users have the potential to unfairly consume shared resources. Furthermore, there is no way for users to place demands on the quality (*e.g.*, performance) of these services. The need for a resource management infrastructure raises four questions:

1. What resources should be managed?
2. How are management policies specified?
3. How are policies enforced?
4. What policies should be used?

Questions 1, 2 and 3 are well studied. Most researchers agree that an effective approach should focus on controlling the physical resources of the network: node CPU time, memory , disk space and network bandwidth. Some work has also been done in policy specification in general [3] and without concrete demonstration for Active Networks [20]. Finally, some projects have examined techniques for enforcing resource usage, including namespace management [1, 12], runtime access control [7], limited expressibility [11, 17], certification [24], and fine-grained resource accounting [14, 6].

We believe that the central outstanding question in effective resource management is question 4, the specification of scalable policies. In this paper, we present a solution to this problem, consisting of two components. At the policy level, we define a distributed, market-based policy for resource allocation. This is in sharp contrast to the more administrative-based policies proposed to date; these policies inhibit interaction throughout the network because they are fundamentally local and proprietary. Instead, a market-based approach potentially 'opens up' the entire network, transforming it into an open service market. At the mechanism level, we integrate KeyNote [3], a distributed *trust-management system*, into the architecture of active elements. KeyNote is roughly similar to Java's SecurityManager [7], in that it is an entity that authorizes decisions of policy, but differs in that policy components may be delegated and distributed as credentials, thereby facilitating greater scalability. KeyNote serves to specify and uphold the 'rule of law' that governs market interactions.

Market-based policies for resource management are not new; they have been applied to bandwidth allocation [16], memory allocation [10], and CPU scheduling [23]. Our approach was inspired by the work of Stratford and Mortier [18], who propose a market-based approach to QoS-oriented resource management for operating systems. In their work, dynamic pricing is used as a mechanism to enable applications to make policy-controlled adaptation decisions. Contracts may be established between users and applications, and between applications and resource managers to apportion resources. These contracts may be established directly, or via third-party 'resource traders'. We apply similar market-based models to the active networking context. While the areas of trust management, market-based systems and active networks are not new, it is their combination that constitutes our novel contribution.

In the remainder of this paper, we present our design of an active network infrastructure that implements market-based resource management. In Section 2 we provide an overview of our approach, elaborating on how market-based control and trust management can materialize a scalable framework for addressing resource management issues in active networks. Section 3 presents the current state of our implementation. Some first experiments and an example application that expose the benefits of our work are described in Section 4. Section 5 provides impressions, conclusions and directions for further investigation.

2 Overview

In order to create a scalable framework for apportioning network resources, we need two things: a scalable policy and a scalable way of expressing and enforcing that policy. For the former, we look to specify a *market-based policy*; for the latter, we use a decentralized *trust-management* system to express and distribute that policy information to the enforcing elements.

In this section we present an overview of our approach, leaving the details to the next section. We begin by explaining the merits of the market approach and then describe how we use it in particular. We finish by describing the benefits of trust management and how it applies to our system.

2.1 Market-based control

A market can be defined as "*... a set of interacting agents with individual goals that achieve a coherent global behavior ...*" [5]. Interaction takes place in the form of buying, selling or trading of goods. Each agent implements a strategy for achieving its goals through the interactions within the market. Economic theory suggests that through the rational behavior of the agents and the competitive process, the market converges to equilibrium where supply and demand match, and near optimal allocations of goods are achieved. In general, agents can be producers of goods, consumers of goods or both. The ones that act as producers are concerned with setting prices in order to maximize profit while consumers try to maximize perceived utility given specific budget constraints. This ability to facilitate fair allocation with very little information (just the price) makes the market an attractive framework for solving many complex problems.

For our purposes, the prime advantage of a market is its lack of centralized control. Instead, control emerges from the individual goals and actions of the agents and is thus inherently distributed among the elements of the market. The decentralization property is crucial in the design of scalable systems: the system is allowed to grow and change without bottleneck.

There are a number of other practical advantages of creating an *active network economy*:

- No assumptions are made on the cooperative or competitive nature of the agents and their goals; there can be as many different strategies as agents, and each strategy may be more or less "selfish" and non-cooperative. In active networks, we also consider selfish and untrusted users to be able to execute network programs thereby consuming resources. The market-based approach offers flexibility in dealing with different degrees of cooperation and competition in the control process.
- A market is abstract enough to capture resource control problems at any scale, from lower layer resources such as CPU and bandwidth to higher layer services such as secure connections, streams with guarantees on performance, *etc.*
- We can build upon this infrastructure to actually charge for network services.

We apply the market model to active network resource management. We define what constitutes the general terms *agent*, *good* and *trade*, as we have described above.

The active network economy In the active network economy, the goods traded are the physical resources of the active nodes, *e.g.,* CPU, memory, network capacity and secondary storage. Typically, the producing agents are the elements of the nodes, and the consuming agents are active programs that wish to run on the nodes and use the nodes' elements. We also define a class of agents, called *service brokers*, for mitigating access between producers and consumers. Service brokers peddle *resource access rights*, the form of currency in our marketplace. A resource access right is essentially a promise of a certain sort of service, *e.g.,* a periodic slice of processing time, or a fixed amount of network bandwidth. Resource access rights are themselves purchased by the users of active programs. Service brokers may manage a number of producers' resources or may in fact be implemented as part of the producer. Also, a single service broker can manage resources from different nodes across the active network, which constitutes a highly valuable feature with regard to our concerns for scalability.

In our implementation, instead of authorizing access to physical resources directly, we authorize access to *functions* that access those resources. Resource access rights generally specify three things: *what* function may be called, *when* it may be called and any *restrictions* on its arguments. For example, rather than issuing a right to some slice of CPU time, we may issue a right to set a certain scheduling priority. This is expressed as a right to call `Thread.set_priority` once with an argument that is less than some number. This approach also allows us to generalize any *active service* as a good in our economy, rather than just raw resources. On the other hand, if the underlying mechanisms are available to enforce access to raw resources directly, as in [14], these rights can also be accommodated.

Resource access rights are part of a more general policy that governs how an active node may be used and by whom. Resource access rights are policy that is sold on the market, and can be combined into a framework that also accommodates administrative policies. In fact, administrative policy might govern what market policies are to be made available under what conditions. The next question is how to integrate these policies with mechanisms used for enforcing them on the active nodes. For this task we have made use of a *trust-management system*, KeyNote, which we describe next.

2.2 Trust Management

Since resource access rights are essentially policy rules governing the use of active network resources, we must establish some basic requirements for the policy scheme we use:

1. Decentralization in specifying (and enforcing) policies and access control rules. This is preferable both to the individual entities that wish to specify

```
KeyNote-Version: 2
Authorizer: CPU_BROKER
Licensees: BOB
Conditions: (an_domain == "an_exec" && module0=="Thread"
             && module1= "set_prio" && arg1 < 30
             && @onetime == "yes") -> "ACCEPT";
Signature: "rsa-md5-hex:f00f5673"
```

Fig. 1. Example credential guaranteeing a user a share of the CPU.

their own policies, as well as to the entire economy, since a central point of enforcement for all transactions would yield a system that does not scale.

2. Flexibility in establishing relationships between entities in the network on varying time-scales. For example, service brokers may change relationships with various producers, and therefore the resource access rights that they sell should reflect the change. Consumers should be able to choose appropriate producers and buy from whomever they wish.

Trust Management is a novel approach to solving the authorization and (security) policy problem, introduced in [4]. Entities in a trust-management system (called *"principals"*) are identified by public keys, and may generate signed policy statements (which are similar in form to public-key certificates) that further delegate and refine the authorization they hold. This results in an inherently decentralized policy system; the system enforcing the policy needs to consider only the relevant policies and delegation credentials, which the user has to provide.

In our system, resource access rights are implemented as policies initially authorized by the resource producer. At the outset, these policies are applicable to the service brokers, who may then delegate (all or part of) them to the consumers who purchase them. Consumers then provide the policy credentials to the producer when they want to access the service.

We have chosen to use KeyNote [3] as our trust management system. KeyNote provides a simple notation for specifying both local policies and credentials. Applications communicate with a "KeyNote evaluator" that interprets KeyNote assertions and returns results to applications. The KeyNote evaluator accepts as input a set of local policy and credential assertions and a set of attributes, called an "action environment" that describes a proposed trusted action associated with a set of public keys (the requesting principals), and finally returns whether proposed actions are consistent with local policy. In our system, we use the action environment to store component-specific information (such as language constructs, resource bounds, *etc.*) and environment variables such as time of day, node name, *etc.*, that are important to the policy management function.

As an example of a KeyNote credential, Figure 1 shows a resource access right for a specific share of the CPU, as we described in the last subsection. The credential indicates that `BOB` may call `Thread.set_prio` once, at most, with the condition that the argument is less than 30.

3 Implementation in ALIEN

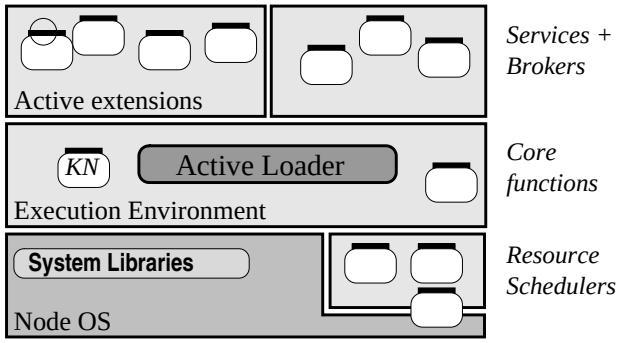


Fig. 2. The ALIEN Architecture and the new components: Keynote (KN Module) and the resource schedulers, which are part of the Execution Environment, and Brokers, which are implemented as active extensions.

So far we have described our approach in general; in this section we describe our implementation, focusing on what policies we use and how they are enforced. The space of allowable policies is inherently tied to that which can be enforced. For example, a policy stating that a user may only use 10% of the CPU over a certain period would be totally useless without a corresponding mechanism to enforce that policy. In this sense, the available enforcement mechanisms establish the *vocabulary* for specifying policies. These enforcement mechanisms are themselves dependent on what is made available in the implementation, which we describe here.

Our implementation builds on the SwitchWare project's ALIEN [1] prototype, whose three-layer archticture is shown in Figure 2. In ALIEN, properly authorized users may extend node processing with new code, termed *active extensions*, using an API defined by the *core functions*. These core functions present an interface to node resources, essentially as an abstraction of the OS. The visible API is controlled by the *Active Loader*; currently, loaded code is allowed a smaller API than statically linked functions enforced at load-time, for security purposes. For example, loaded code does not have access to the local disk.

We have extended ALIEN to provide enforcement for our market-based policies, in two ways. First, we have modified the Active Loader to control the visible API of active extensions in a more fine-grained manner. Extensions may either have full access to a particular core function, partial access (that is, as long as certain parameters are within certain limits) or no access. This decision is made on a per-extension basis, according to policy. Access control is enforced at load-time when possible, or else at run-time. Second, we have exposed some of the functionality of the *resource schedulers* into the core functionality, so they may

be similarly controlled by the Active Loader. Resource schedulers control low level functions, such as thread-scheduling, packet-scheduling, etc.

For the remainder of this section, we describe these two extensions to ALIEN. Then we conclude with some of the details of how we implement Service Brokers.

3.1 Controlling Active Extensions

In ALIEN, active packets or extensions are received and handled by the Active Loader, which manages the active node execution environment. It is the Active Loader's task to instantiate the code received by dynamically linking it to the environment and, if necessary, create a new thread for executing it.

We extended the dynamic linking process to perform policy compliance checks on the module references and trigger one of three actions: accept or reject the reference, indicate that further run-time checking on the reference arguments is needed or initiate a policy-based replacement of a reference with a more appropriate one. This last feature provides a very useful technique for translating generic references to specific service implementations or service variants, according to policy. It can be used, for example, to select between different connection oriented communication service stacks (Secure *vs.* Unsecure) or choose between guaranteed or best effort service (by translating references to the `Thread` module to either `BEThread` for best-effort or `GSThread` for guaranteed service).

This policy enforcement mechanism is implemented as follows. At dynamic link time, the linker processes the bytecode object and extracts all references to modules and functions that are external to the loaded object. For each such reference, the KeyNote evaluator is queried, along with any appropriate credentials. Each result type is associated with a specific action that has to be taken. The currently implemented result types are:

- `ACCEPT`: the reference is accepted unconditionally.
- `REJECT`: the reference is not allowed.
- `REPLACE`: the reference has to be replaced by another reference; *e.g.*, a reference to a general packet-sending function is replaced by a rate-limited one.
- `CHECK-ARGS`: there are restrictions on the arguments of the reference.

For the first three cases, the linker can perform the check and take necessary actions, resulting in no run-time penalty. For the final case, checks must occur at run-time, since function arguments are dynamic entities. Each such reference is replaced by an anonymous function. This function contains instructions to initiate a query to the KeyNote evaluator on the validity of the original functions' arguments. An `ACCEPT/REJECT` reply is expected. A `REJECT` reply would raise an `Invalid_Argument` exception, while `ACCEPT` would result in the anonymous function calling the original reference. These interactions with the policy management element are shown in Figure 3.

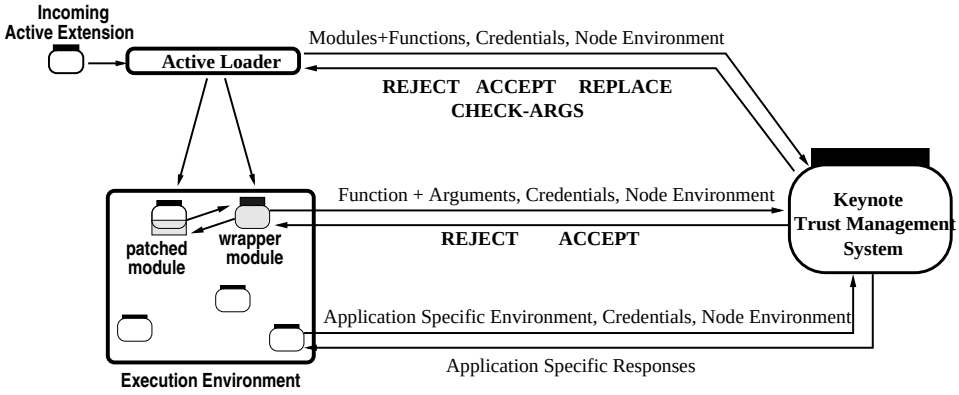


Fig. 3. Interactions with the Keynote Trust Management System: Link-time, Run-time, and application specific interactions are depicted.

3.2 Resource Schedulers

The task of resource schedulers is to coordinate access and control allocation of shared resources such as CPU, memory, storage or network bandwidth. The ALIEN implementation includes support for thread scheduling and memory allocation as part of the Active Loader. Packet scheduling can be dealt with in a similar fashion; however, since ALIEN is currently implemented in user-space and relies on the operating system for scheduling packets, we do not address such issues in this paper. We focus on the use of a market-based hierarchical thread scheduler, and its interaction with the policy management and service broker functions.

Hierarchical scheduling enables the coexistence of different scheduling disciplines to fit the various application requirements. In practice, clients of the scheduling process can be schedulers themselves, thus forming a scheduling tree. It is also possible for users to attach their own schedulers to the hierarchy in order to manage some share of the processing power.

At the root of the hierarchy, we use a market-based algorithm called lottery scheduling [23]. In lottery scheduling, the notion of *tickets* [22] is introduced. These are data structures that encapsulate resource access rights. Each ticket is expressed in a certain *currency* that is significant to the issuer and the holders. In the case of lottery scheduling, tickets represent a share of the available processing power. At each scheduling interval, tickets participate in a drawing. Each ticket's probability of being selected is proportional to its share value. When a ticket is selected, its owner is scheduled for execution[1].

We implemented two kinds of second-level schedulers to provide best-effort and guaranteed-share service disciplines. They are both based on lottery schedul-

[1] One anonymous reviewer of non-technical background commented to this: "Remind me not to buy one of your lottery tickets!"

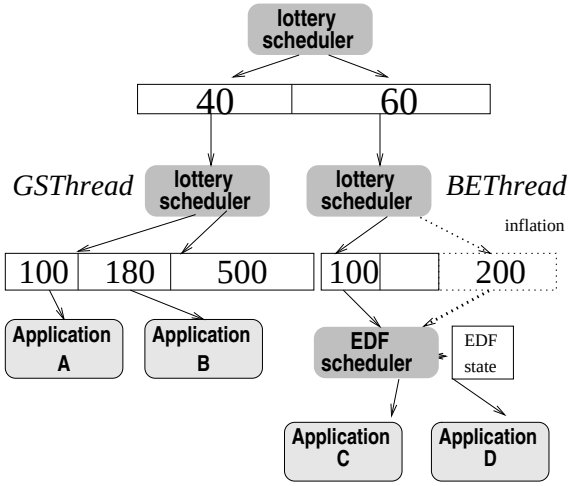


Fig. 4. A typical scheduling hierarchy.

ing but differ in the way they manage their tickets. The guaranteed-share scheduler has to maintain a constant sum of ticket values that corresponds to a specific share of the processing power. It may issue tickets to its client threads, but the sum of the ticket values has to be conserved. In contrast to that, the best-effort scheduler might continue to issue new tickets without maintaining a constant sum. The new tickets cause inflation, the share value of each ticket drops, and so does performance of its owning thread. It is possible to implement price-adaptive scheduling under the best-effort scheduler by monitoring performance and buying more tickets according to a utility function. A deadline-based scheduler can also be built under the top-level lottery scheduler: its task would be to acquire and release tickets according to its current demand for meeting deadlines. An example scheduler hierarchy in our system is shown in Figure 4.

The ALIEN execution environment provides an interface to the built-in scheduler in the `Thread` module. For the lottery scheduler, we retained the same interface, with one new function added, `set_ticket`, which sets the tickets of the current thread to a specific type and value. Credentials may control how this function is called and with what arguments. However, it is the scheduler's task to interpret the `set_ticket` call according to its own semantics. The call might simply cause inflation, trigger a call to acquire more resources from the parent scheduler or be rejected. We implemented the lottery scheduler so that it is customizable and reusable. Under appropriate circumstances, users may override default functionality and implement the ticket management functions themselves.

3.3 Service Brokers

Service brokers provide the necessary credentials for enabling a service, and encapsulate service management functions. Conceptually, this function is equivalent to the trading function as described in [15, 13, 8, 9]. The Internet Diffserv architecture [2] also considers the Bandwidth Broker [21, 19] for managing network bandwidth. From the policy perspective, brokers are principals to which authority is delegated for managing a specific policy subspace. The implementation issues here are how the broker that is responsible for a service is identified and how users can communicate with brokers.

Since authority over some policy subspace is delegated to some broker, there exists information in the policy management system — in the form of credentials — that indicates this delegation. This information can be obtained by users and, through this service-principal-broker association, users can then establish communication with the broker. We implemented a broker communication module (BCM) whose task is to relay communication between brokers and users. The BCM allows users to find the appropriate broker (among those that have registered with the BCM) for the resources they need to consume, query those brokers on the availability and price of various services and acquire such services by providing the appropriate credentials.

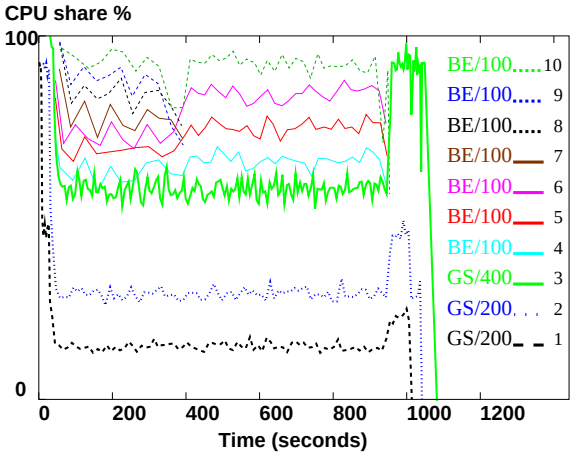
4 System Demonstration

In this section we share some first experiences with our resource management system. First, we focus on the market-based CPU scheduler and demonstrate its ability to meet arbitrary user demands in various situations. We then describe the implementation of an active web proxy that exploits all different aspects of our architecture to their full extent.

4.1 Evaluation of the market-based scheduler

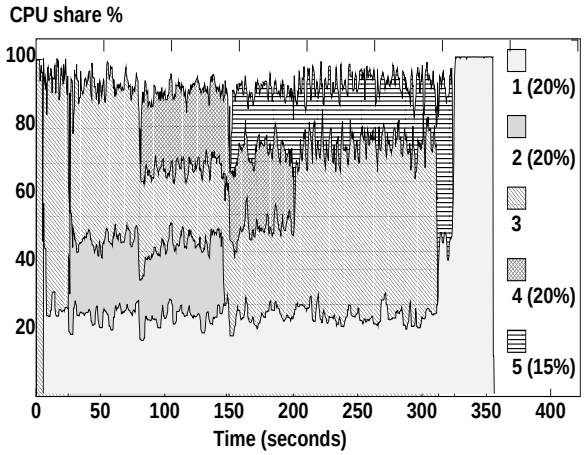
In this experiment, a number of threads execute on the active node and consume CPU cycles. At each thread's initialization, it is assigned a fixed amount of tickets for a specific scheduling class. Since we implemented two service classes, GSThread and BEThread, their schedulers hold tickets of the root scheduler: GSThread has allocated 500 and BEThread 300, which notes a 5:3 ratio of CPU resource shares between these two classes. The starting times, service class and ticket amount for each thread in our scenario and the consumption of CPU cycles over time for all threads are shown in Figure 5.

We observe that the three guaranteed service threads are successfully isolated from the other threads that arrive over time. Also, after the best-effort class becomes empty, its share also becomes available. Note that thread 3 takes double the share that 1 and 2 have, exactly as defined by their ticket values. Note also that fairness is preserved in the best-effort class, in that all threads receive equal share of service since they have equal ticket values.



Start	ID	Type	Tickets	Load
0.00	1	GS	200	100
8.85	2	GS	200	100
30.89	3	GS	400	400
43.98	4,5,6	BE	100	40
	7,8,9			
55.98	10	BE	100	40

Fig. 5. CPU resource consumption for a number of different threads. Thread characteristics are shown in the table, where *Load* is the amount of CPU time the threads need to complete.



Start	ID	Type	Load	Goal
5.0	1	BE	300	20 %
25.0	2	BE	80	20 %
0.0	3	BE	360	-
80.0	4	BE	80	20 %
150.0	5	BE	100	15 %

Fig. 6. CPU resource consumption for the second experiment; characteristics are shown in the table, where *Goal* is the thread's target processing share while adapting to the system's changing conditions. The threads that dynamically adjust their ticket values are clearly visible.

This simple experiment demonstrates thread isolation and differential service quality. However, no interactions take place between schedulers and threads, except for acquiring tickets at thread creation time. The next experiment features threads that dynamically acquire or release tickets according to a specific goal. Threads wish to maintain constant performance, while being clients of the best-effort service class. Periodically the threads evaluate their performance (processing rate) and readjust their ticket allocation to reflect their goal. Other threads that have no such requirements simply acquire tickets at creation time only. The characteristics of threads in this experiment and the threads' CPU over time are shown in Figure 6.

One can clearly see the ones that try to adapt to the environment. Figure 7 shows the number of tickets for each thread. More elaborate strategies can be used that consider budget constraints and utility. It is also possible to allocate a GSThread ticket share as a secured minimum performance and BEThread tickets for further performance gains. Through these two simple experiments we give some insight into the flexibility, scalability, and applicability of our system, offered by a market-based approach to lower-level resource scheduling. In the next section, instead of being focused on a specific function of our system we utilize all the components we implemented.

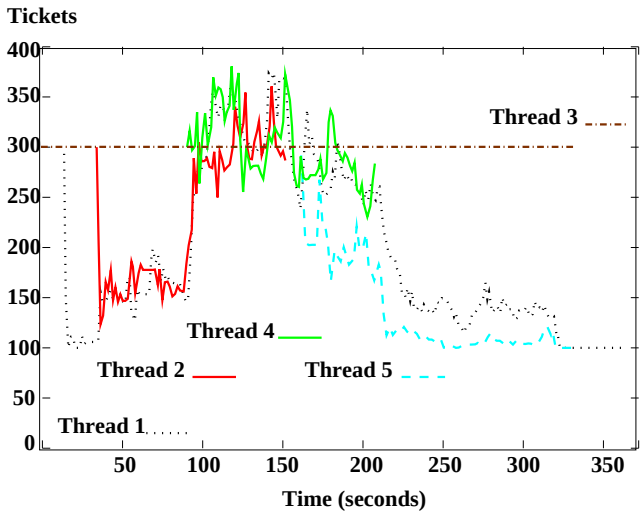


Fig. 7. Ticket allocation over time for the 4 threads that dynamically allocate CPU power.

4.2 The active web proxy

To validate our system design and evaluate its impact, we implemented a web proxy server as an active extension that is loaded by users on active nodes. The following characteristics of this application are essential:

- The user must be authorized to install the active extension, and the active extension should be able to access the modules and functions it needs. Such authority is granted to the user by some administrator, or by interaction with the market, for acquiring credentials. Users must then provide the necessary credentials for the web proxy to be able to act on their behalf, *i.e.,* with their own resource access rights.
- Network-level access control is needed to indicate which server addresses the proxy is allowed to connect to as well as which client addresses are allowed to connect to the proxy. In the active node's local policy, the `Tcp` module is mapped to the `Kntcp` module (through a `REPLACE` credential), a wrapper module to `Tcp`. `Kntcp` is implemented so that every connection is subject to access checks. This could also be done using a `CHECK-ARGS` policy to check the arguments of the `connect`, `accept` or `bind` functions. Credentials supplied by the user authorize the proxy to "talk to" certain addresses.
- CPU resources are consumed and should therefore be controlled through the market-based scheduler and the service broker function. Determining the appropriate brokers and credentials to use is implemented in the exception handlers that deal with failures during linking. These handlers contact the `BCM` module to determine the relevant available services and costs, and acquire the relevant credentials (which authorize the proxy to link with the selected thread-scheduling service) from the selected broker. The linking process is then restarted with the newly-acquired credentials until another exception is raised or linking is successfully completed.

The active web proxy can then proceed to service requests. While this experiment proves our concept, its development provided a few interesting observations. First, directly writing a large number of KeyNote credentials might not be the easiest way to specify policy. A higher-level language that can then be compiled into a number of KeyNote credentials could be more useful. Second, a potential performance bottleneck that could negatively affect scalability is the instant creation of credentials by the service brokers. On a 500 MHz Pentium III processor, signing a credential takes $8msec$ which corresponds to 124.36 credentials/second[2]. There are methods to overcome this limitation such as having pre-issued credentials, using lighter cryptographic algorithms or employing hardware support.

5 Conclusions

We have addressed the problem of scalable resource management in active networking and , based on the scalability of market-based mechanisms, developed

[2] Verifying a credential is much faster, at $10.71\mu sec$, or 93457.94 credentials/second.

a novel system architecture for allocating and adjusting resource allocations in a system of communicating active network nodes. We have used a trust-management policy system, KeyNote, which allows us to ensure that resource allocations are controlled and enforced under specified policy constraints. We believe that the resulting system is the first system which has provided a network-level resource management framework for active networking, moving beyond the node architecture considerations which have occupied much of the design efforts in first-generation active networking research.

We believe that the system architecture described herein has considerable applications outside active networking. For example, it might serve as an equally powerful resource management paradigm in inter-networks where RSVP or other integrated services notions are used to control resources. While we have focused on active networking as our immediate concern, we intend to investigate the applicability of this system architecture to a wider set of distributed resource management problems. We believe that the scalability and security of this system are powerful attractions and that these fundamentals can be preserved across many changes of the environment.

Acknowledgements

This work was supported by DARPA under Contracts F39502-99-1-0512-MOD P0001 and N66001-96-C-852. We thank the members of the Distributed System Laboratory and the reviewers for their helpful comments and fruitful discussions before and during the course of this work.

References

[1] D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks.* PhD thesis, University of Pennsylvania, September 1998.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. Technical report, IETF RFC 2475, December 1998.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust management system version 2. Internet RFC 2704, September 1999.

[4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.

[5] Scott Clearwater. Why market-based control? In *Scott Clearwater, Ed. , Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages v–xi. World Scientific Publishing, 1996.

[6] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of the 1998 ACM OOPSLA Conference, Vancouver, BC*, October 1998.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, Reading, 1996.

[8] Object Management Group. *A Discussion of the Object Management Architecture.* January 1997.

[9] Object Management Group. Trading Object Service Specification. In *OMG Doc 97-12-23*, March 1997.

[10] Kieran Harty and David Cheriton. A Market Approach to Operating System Memory Allocation. In *Scott Clearwater, Ed. , Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages 126–155. World Scientific Publishing, 1996.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[12] Michael Hicks and Angelos D. Keromytis. A Secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

[13] E. Kovacs and S. Wirag. Trading and Distributed Application Management: An Integrated Approach. In *Proceedings of the 5th IFIP/IEEE International Workshop on Distributed Systems: Operation and Management*, October 1994.

[14] Paul Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *Proc. of the 1st International Working Conference on Active Networks*, June 1999.

[15] M. Merz, K. Moeller, and W. Lamersdorf. Service Trading and Mediation in Distributed Computing Systems. In *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 450–457, 1994.

[16] Mark S. Miller, David Krieger, Norman Hardy, Chris Hibbert, and E. Dean Tribble. An Automated Auction in ATM Network Bandwidth. In *Scott Clearwater, Ed. , Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages 96–125. World Scientific Publishing, 1996.

[17] Jonathan T. Moore. Safe and Efficient Active Packets. Technical Report MS-CIS-99-24, Computer and Information Science, The University of Penns lyvania, 1999.

[18] Richard Mortier Neil Stratford. An Economic Approach to Adaptive Resource Management. In *Proc. of Hot topics in Operating Systems*, 1999.

[19] K. Nicols, V. Jacobson, and L. Zhang. A Two Bit Differentiated Services Architecture for the Internet. Internet Draft, November 1998.

[20] Morris Sloman and Emil Lupu. Policy Specification for Programmable Networks. In *International Working Conference on Active Networks (IWAN)*, 1999.

[21] Andreas Terzis, Jun Ogawa, Sonia Tsui, Lan Wang, and Lixia Zhang. A Prototype Implementation of the Two-Tier Architecture for Differentiated Services. In *Fifth IEEE Real-Time Technology and Applications Symposium, Vancouver, BC, Canada*, June 1999.

[22] C. A. Waldspurger and W. E. Weihl. An Object-Oriented Framework for Modular Resource Management. In *Proc. of the 5th International Workshop on Object Orientation in Operating Systems*, pages 138–143, October 1996.

[23] C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional Share resource management. In *Proc. of the First Symposium on Operating System Design and Implementation*, pages 1–11, November 1994.

[24] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-based System. In *Proceedings of the 17th ACM Symposium on Operating System Principles, Kiawah Island, SC*, December 1999.