RCANE: A Resource Controlled Framework for Active Network Services

Paul Menage

University of Cambridge Computer Laboratory
Pembroke Street, Cambridge, CB2 3QG, UK
Paul.Menage@cl.cam.ac.uk

Abstract. Existing research into active networking has addressed the design and evaluation of programming environments. Testbeds have been implemented on traditional operating systems, deferring issues regarding resource control. This paper describes the architecture, resource models and prototype implementation of the Resource Controlled Active Network Environment (RCANE). RCANE supports an active network programming model over the Nemesis Operating System, providing robust control and accounting of system resources, including CPU and I/O scheduling, and garbage collection overhead. It is thus resistant to many classes of denial of service (DoS) attack.

1 Introduction

Adding programmability to a network greatly increases its flexibility. However, with this flexibility comes greater complexity in the ways that network resources, including CPU, memory and bandwidth, may be consumed by end-users. In a traditional network, the resources consumed by an end-user at a network node are roughly bounded by the bandwidth between that node and the user; in most cases, the buffer memory and output link time consumed in storing and forwarding a packet are proportional to its size, and the CPU time required is likely to be roughly constant. Thus, limiting the bandwidth available to a user also limits the usage of other resources on the node.

In an active network hostile (or greedy or careless) forwarding code could potentially consume all available resources at a node. Even in the absence of specific denial of service (DoS) attacks, the task of allocating resources according to a specified Quality of Service (QoS) policy is complicated by lack of knowledge about the behaviour of the user-supplied code.

The resources consumed by untrusted code need to be controlled in two ways. The first is to limit the execution qualitatively – limit *what* the code can do. This involves restricting either the language in which the code can be written, or the (possibly privileged) services which it can invoke. The second way is to limit the code quantitatively – limit *how much* effect its activities can have on the resources available to the system. This requires fine-grained scheduling and accounting.

Stefan Covaci (Ed.): IWAN'99, LNCS 1653, pp. 25–37, 1999.

This paper discusses the design and implementation of a framework to permit such quantitative and qualitative control. Section 2 outlines the design of the framework. Section 3 discusses its implementation on the Nemesis Operating System. Section 4 presents experiments to demonstrate the validity of the approach. Section 5 surveys related approaches to active networking and resource control.

2 RCANE Design

This section provides an outline of the design of RCANE. The architecture follows the principles given in [1] to partition the system:

- The *Runtime* is written in native code and provides access to, and scheduling for, the resources on the node, and services such as garbage collection (GC).
- The *Loader* is written in a safe language (OCaml [2] in the current implementation of RCANE), as are all higher levels. The Loader is responsible for system initialisation and loading/linking other code.
- The *Core*, loaded at system initialisation time, provides safe access to the Runtime and the Loader and performs admission control for the resources on the node.
- *Libraries* may be loaded both at system initialisation and by the actions of remote applications. They have no direct access to the Runtime or the Loader, except where permitted by the Core.

2.1 Discussion – Hardware or Software Protection?

Since an active network node is expected to execute untrusted code, there needs to exist a layer of protection between each principal and the node, and between principals. It is possible to utilise the memory protection capabilities of the node's hardware, allowing principals to execute programs written in arbitrary languages [3]. However, at the time-scales over which active network applications are likely to execute, this paradigm is too heavyweight. An alternative, taken by RCANE, is to require principals' code to be written in a safe, verifiable language. This allows much of the protection checking to be done statically at compile or load time, and allows much lighter-weight barriers between principals. In particular, it means that interactions between principals can be almost as efficient as a direct procedure call.

2.2 Sessions

A *Session* represents a principal with resources reserved on the node. Sessions are isolated, so that activity occurring in one session should have no effect on the QoS received by other sessions, except where explicit interaction is requested (e.g. due to one session using services provided by another session).

Figure 1 shows part of the `Session` interface provided by the Core to permit control over a session and its resources. `createSession()` requests the creation

of a new session. Credentials to authenticate the owner of the new session for both security and accounting purposes are supplied, along with a specification of the required resources and the code to be executed to initialise the session. `destroySession()` releases any resources associated with the current session. `loadModule()` requests that a supplied code module be loaded and linked for the session. `linkModule()` requests that an existing code module (possibly loaded by a different session) be made available for use by this session. The code may be specified simply by the interface which it exports, or by a digest of the code implementing the module, to prevent module-spoofing attacks. `bindDevice()` reserves bandwidth and buffers on the specified network device. Other functions concerning modification of resource requirements are not shown.

```
bool createSession (c : Credentials, r : ResourceSpec, code : CodeSpec);
void destroySession (void);
bool loadModule (l : LoadRequest);
bool linkModule (l : LinkRequest);
bool bindDevice (d : Device, bu : BufferSpec, bw : BandwidthSpec);
```

Fig. 1. Part of the `Session` interface

At system initialisation time two sessions are created:

- The System session represents activity carried out as housekeeping work for RCANE. It has full control over the Runtime. Many of the control-path services exported from the Loader and the Core are accessed through communication with the System session.
- The Best-Effort session represents activity carried out by all remote principals without resource reservations. Packets processed by the Best-Effort session supply code written in a restricted language and are given minimal access to system resources. Access to `createSession()` is permitted, to allow code to initiate a new session; further packets may then be processed by the newly created session.

2.3 Resource Accounting

Resources used by sessions running on RCANE are be accounted to the appropriate session, and charged to the principal who authorised the creation of the session. Pricing and charging policies will be system dependent.

Resource requests are processed by the System session and, if accepted, are communicated to the Runtime's schedulers. In general, data-path activity, e.g. sending packets, is carried out within the originating session. System modules in the Core are linked against entry points in the (unsafe) Runtime; these are then exported through safe interfaces to which the untrusted sessions can link directly. The Runtime performs a policing function on use of the node's resources.

2.4 CPU Scheduling

RCANE uses the following abstractions to control CPU usage by sessions:

- A *virtual processor* (VP[1]) represents a regular guaranteed allocation of CPU time, according to some scheduling policy. A session may have one or more VPs. All activities carried out within a single VP share that VP's CPU guarantee.
- A *thread* is the basic unit of execution, and at any time is be either *runnable* (working on computation), *blocked* (e.g. on a semaphore, or awaiting more resources to become available) or *idle* (in a quiescent state, awaiting the arrival of further work items).
- A *thread pool* is a collection of one or more threads. Each thread is a member of one pool. Associated with each pool is a queue of packets and a queue of events. Each pool is associated with a single VP; its threads are only eligible to run when its VP receives CPU time.

Incoming packets (see Sect. 2.5) are routed to the associated pool and added to its packet queue. *Events* (functions for execution at a given time in the future) may be added to a pool's event queue. Whenever there is work to be done in a pool (either newly arrived packets, or events whose timeouts have passed), any idle threads in the pool are be dispatched to process the work. When a running thread has finished its task, it returns to the idle state.

This allows sessions flexibility in how they map their work onto threads. For tasks that have to be processed serially (e.g. routing a stream of packets) a single thread might be bound into a pool, to perform all processing required for that pool. For network services where it is desirable to service multiple requests at a time, several threads can be bound into a single pool, and as packets come in they will be dispatched to an idle thread. Alternatively, sessions wishing to perform both event-driven and packet-driven activities can choose between running two threads in separate pools (to prevent interference between the two activities), or saving resources by having a single thread in one pool (but risking occasional interference between packet and event activity). Similarly, for even better isolation between activities, the session could associate each activity with a separate VP (i.e. give each activity its own particular CPU guarantee).

2.5 Network I/O

Sessions running under RCANE can pass demultiplexing specifications to the Runtime, associating incoming flows of packets with specified pools and processing functions.

To prevent crosstalk between the network activity of different principals, all packets are demultiplexed to their receiving pools by the Runtime at the lowest possible level. As little work as possible is carried out on those packets before

[1] For those familiar with the Nemesis Operating System, over which this work is based, this abstraction is distinct from the normal Nemesis notion of a VP

demultiplexing. Once the VP associated with the receiving pool is given CPU time, one of the pool's idle threads can be used to invoke the flow's processing functions.

This allows each session full control over decisions such as whether, and what kind of, authentication is used for packets on a given flow. For non-authenticated flows, a session can specify a function which processes the packet's payload immediately; should authentication be required, the session's favoured authentication routines may be invoked with the relevant authentication data from the packet.

A session may request a guaranteed allocation of buffers for receiving packets from a given network device. Incoming packets demultiplexed to the session will be accounted to this allocation, and returned to it when packet processing is completed. Packets for sessions without a guaranteed allocation are received into buffers associated with the Best-Effort session. Thus, although such sessions can receive packets, they will be competing with other sessions on the node.

Similarly, a session may request its own allocation of guaranteed transmission bandwidth and buffers for a specified network device, or may use the transmission resources of the Best-Effort session.

2.6 Memory

The memory managed by RCANE falls into four categories: network buffers (discussed in Sect. 2.5), thread stacks, dynamically-loaded code and heap memory. Network buffers and thread stacks are accounted to the owning session in proportion to the memory consumed. Charging for keeping code modules in memory is likely to be a system specific policy e.g. it might be the case that linking to a commonly used module would be less expensive than loading a private module.

Heap memory presents more challenges. Since safe languages generally require GC to prevent malicious (or careless) programmers exploiting dangling pointers, RCANE needs to provide GC services. The framework must be able to support the following features:

- Efficient tracking of the memory usage of each session.
- Ability to revoke references from other sessions when a session is deleted.
- Prevention of crosstalk between sessions due to GC activity.

These three requirements suggest giving each session its own independently garbage-collected heap. Tracking the allocations made by a session is straightforward; deciding to whom to refund garbage-collected memory is difficult to perform efficiently without separate heaps. Sessions which have completed their tasks (or whose authorisation/credit has expired) are destroyed – if other sessions have pointers to their data, it is impossible to safely release the session's memory. Finally, deciding to whom to account the time spent on GC activity is difficult without separate heaps.

RCANE uses an incremental garbage collector to prevent excessive interruptions to execution. Each session reserves a maximum heap size, and tunes the parameters of the GC activity – such as frequency and duration of collection

slices – to allow it to trade off responsiveness against overhead. Charging can then be based on the size of the reserved memory blocks that comprise the heap, rather than the amount of live memory within those blocks, simplifying the accounting process.

2.7 Service Functions

The use of separate heaps and garbage collectors for each session requires RCANE to prevent the existence of pointers between different sessions' heaps. In general this does not present a problem, since applications will not generally be relying on shared servers to perform data-path activities. However, in some situations it may be necessary or desirable to communicate with other sessions:

- When talking to the System session to request a change in reserved resources, or to make use of services provided by the System session (such as default routing tables).
- Some sessions may wish to export services to other sessions running on the node (e.g. extended routing tables, or access to proprietary algorithms).

In each of these cases, a client executing in one session requires a local reference to a *service* function implemented in a different session. This reference is opaque to the client, and enables the runtime to identify the service associated with the reference. Invoking this service involves the following steps:

1. Copying the function's parameters into the server session's heap.
2. Invoking the underlying function in the context of the server's heap.
3. Copying the results back into the client session's heap.

During both the invocation and return copying phases, the runtime notes when a copied value is itself a service, and creates a new reference (or reuses an existing reference) to the same service which is available in the destination session. Services can thus be passed from session to session. Server-specified policy can limit such copying to allow additional control over which sessions can utilise a service. Any work carried out by the server during the invocation is performed using the client's thread, and accounted to the client's CPU allocation.

Figure 2 shows the interface provided for creation and manipulation of services. `create()` takes an ordinary function and returns a service function – invoking the returned function will cause the session switch described above. Thus invoking a service appears the same as invoking an ordinary function. Other parameters to `create()` specify the maximum amount of memory to be copied when invoking the service and whether the service may be passed from one client to another. The memory limit is currently a rather crude method of preventing DoS attacks by clients on servers. Ideally, the server would be able to inspect the data before it was copied, but this could result in untracked pointers from the server's heap to the uncopied data in the client's heap. `destroy()` withdraws a service – clients attempting to invoke it in future will experience a `Revoked` exception.

```
type α → β service
exception Revoked
α → β service create (func : α → β; limit : int, shared : bool);
void destroy (s : α → β service);
```

Fig. 2. The `Service` interface

3 Implementation

A prototype of RCANE has been implemented over the Nemesis Operating System [4]. The Runtime is based on the OCaml system from INRIA [2], with support for real-time CPU scheduling, multiple isolated heaps and access to Nemesis I/O. The Best-Effort session uses the PLAN interpreter [5] to provide a limited execution environment for unauthenticated packets, with PLAN wrappers around the `Session` interface to permit authentication and session creation. RCANE interoperates with PLAN systems running on standard (non resource-controlled) platforms, allow straightforward control of an RCANE system. Support for demand-loaded code in the style of ANTS [6] is also provided.

In general, data path operations such as network I/O and CPU scheduling are implemented in native code in the runtime for efficiency. Most control path operations (such as bytecode loading and session creation) are implemented in OCaml for flexibility and ease of interaction with clients.

3.1 CPU Scheduling

CPU scheduling is accomplished using a modified EDF [7] algorithm similar to that described in [8]. Each VP's guarantee is expressed as a slice of time and a period over which the time should be received (e.g. 300μs of CPU time in each 40ms period). Whenever the RCANE scheduler is entered, the following sequence of events occurs:

1. If there was a previously running VP, the elapsed time since the last reschedule is accounted to it.
2. The next VP to be run, and the period until its next pre-emption are calculated. From this point onwards, all work carried out is on behalf of the new VP, and hence can be accounted to it.
3. If there are packets waiting on the owning session's incoming channels (see Sect. 3.3) they are retrieved and transferred to the appropriate pool's packet queues. Any idle pools with pending events are marked as runnable.
4. The next pool and thread to be run are selected.
5. If the selected thread is active in a heap that is currently in a critical GC phase (see Sect. 3.2) then the thread carrying out the critical GC is activated instead, until the phase has completed.
6. The selected thread is resumed.

3.2 Memory

The garbage-collector is based on the OCaml collector. When tracing the roots of a heap, it is necessary to suspend all threads that might access that heap. To ensure that all appropriate threads are stopped during such *critical* GC activity, each thread has associated with it a stack of heaps. When a thread makes a service call through to a different session, a pointer to the server's heap is pushed on to the thread's heap stack. When returning, the server's heap pointer is popped from the heap stack. The top heap pointer on each stack is the thread's *active* heap. (For brief periods of time, while transferring control between two sessions, a thread will actually have both of the top two heaps marked as active.) Whenever critical activity is being carried out on a heap, all threads which are active in the heap are suspended, other than to carry out the GC work. The majority of the GC work can be carried out without suspending threads.

Tracking the threads which have access to each heap minimises the number of threads' stacks which must be traversed to identify roots, and prevents QoS crosstalk between principals which are not interacting. Additionally, a thread executing a service call in a different session need not be interrupted (possibly whilst holding important server resources) due to critical GC work in its own heap. Since no pointers to the client's heap can be carried through to the server, the code running in the server cannot cannot access that heap, and so the thread need not be suspended. Upon returning from the service call it is suspended if the activity is still in progress. When a session is destroyed, any references to its exported services are marked as revoked; attempts to invoke them generate an exception.

3.3 Network I/O

Rcane flows map directly to Nemesis I/O channels. A channel is a connection to a device driver associated with a particular set of flows, specified by a packet filter. The current implementation of Rcane supports channels for UDP packets and Ethernet frames, allowing interaction both on a local physical active network or on a larger virtual network tunnelled over UDP/IP.

Link-level frames are classified on reception by the network device drivers via a packet filter, which maps the frame to the appropriate channel. In the case of sessions without guaranteed resources allocated on a given device, the frame is mapped to the Best-Effort session's channel for that device. If the channel has free buffers available, the frame is placed in the channel – no protocol processing is performed at this point. If the channel has no free buffers, the packet is dropped. Thus, if a session is not keeping up with incoming traffic, its packets will get discarded in the device driver, rather than queueing up within a network stack as might happen in a traditional kernel-based OS.

At some later point, when the appropriate VP is scheduled by Rcane to receive CPU time, the packets are extracted from the channels and demultiplexed to the appropriate thread pools for processing. Transmit scheduling is performed by the device drivers following a modified EDF algorithm, as described in [9].

4 Evaluation

This section presents the results of various test scenarios run to verify the QoS guarantees and resource isolation provided by RCANE.

4.1 CPU Isolation

To demonstrate the isolation of multiple VPs from one another, three separate sessions, each with a single VP, were started at 5 second intervals. In each case an OCaml bytecode module was loaded over the network to be used as the entry point for the session. Session A runs on best-effort time only. Session B requests a 1ms slice in each 4ms period. Session C begins running on best-effort time. After 3 seconds it requests a CPU guarantee of 400μs each 2ms. It makes further changes to its allocation and then exits.

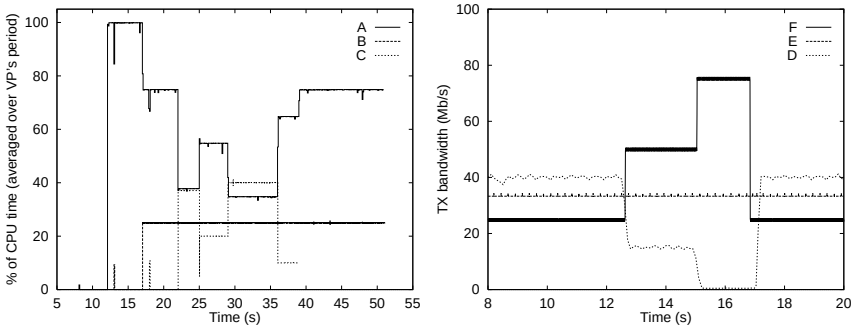
For this experiment, requests for guaranteed CPU allocation also specified that they did not wish to additionally receive a portion of the best-effort time. Figure 3 (a) shows the amount of CPU time actually received by each session over the course of each scheduler period. Initially session A receives all the CPU; later B arrives and receives a constant 25% of the CPU. When C arrives, it initially shares the remaining 75% best-effort time with A; then it switches to guaranteed CPU time (initially 20%, then 40%, then 10%). It can be seen that the guarantees requested from the system were accurately respected at fine timescales.

4.2 Network Transmission

Figure 3 (b) shows a trace of network output from three sessions, each attempting to transmit flat-out. Session D has no guaranteed bandwidth. E has an allocation of 33% (on a 100Mb/s link). F starts with a guarantee of 25%. After about 12s, it requests 45%, thus reducing the best-effort bandwidth available to D. After another 2s, it requests 65%. Now the link is saturated and there is no best-effort transmission time available. After a further 2s it returns to 25%, allowing D to begin transmitting again. It can be seen from the trace that the desired resource isolation is achieved.

4.3 Memory Isolation

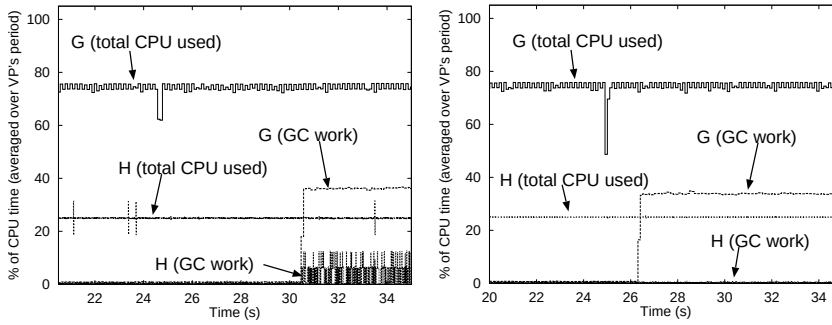
To demonstrate the utility of running different principals' sessions in their own heaps, two scenarios were considered. In (a), VPs G and H are running in the same session. Initially both are generating small amounts of garbage. After a period of time, G begins generating large amounts of garbage. Scenario (b) is the same, but with the two VPs running in separate sessions (and hence having separate heaps). Figure 4 shows the outcome of these scenarios. In (a), both VPs are initially doing small amounts of GC work. G is running best-effort, H has a guarantee of 1ms in each 4ms period. When G switches to generating large



(a) (b)

Fig. 3. (a) Dynamically changing CPU guarantees. (b) Network output

amounts of garbage, the time it spends garbage collecting increases substantially. However, as shown by the noisy region at the bottom right of the graph H also ends up doing an irregular but substantial amount of GC work. Although H has its own independent CPU guarantee, sometimes critical GC activity (such as root tracing) is taking place when its thread is due to run; it must complete this GC work before normal execution can be resumed. In (b), H is unaffected by the extra GC activity caused by G, since it is running in a separate session and hence does not share its heap.



(a) Single heap (b) Separate heaps

Fig. 4. Avoiding QoS crosstalk due to garbage collection

5 Related Work

5.1 Active Networks

Many approaches to loading user-supplied code onto a network node are built on an existing safe language such as Java (including ANTS [6] and Hollowman [10]) or OCaml [2] (including ALIEN [11] and the PLANet service loader [12]). An

alternative is to start with a very restricted language, and rely on the limitations of the language to bound the resources consumed by the user's code. This approach is taken by PLAN [5] and Smart Packets [13]. PLAN also extends the concept of the hop count found in IP to apply to recursive or remote invocations, bounding the resources that a packet can consume globally. The Active Networks Working Group NodeOS Interface Specification [14] aims to standardise on an API addressing similar issues to RCANE, although at a lower level of abstraction.

5.2 Resource Control and Isolation in Safe Languages

JRes [15] provides Java resource control using minimal runtime support. This provides portability, but with high accounting overheads. The J-Kernel [16] gives Java support for multiple protection domains and capabilities to allow revocation of services, but does not fully partition the JVM heap. The Java Sandboxes [17] project allows separate heaps in a modified JVM, by preventing stores of inter-heap references at run-time. This has serious efficiency consequences, and also fails to address the issue of QoS crosstalk due to critical GC activity.

5.3 Resource Control in Operating Systems

Nemesis [4] aims to provide reliable resource guarantees to applications. It is based on the principles that applications should perform as much of their own work as possible, without relying on shared servers for data-path activities, and that applications should have full control over their own resources. The Exokernel [18] takes a similar approach, but motivated by performance gains, rather than provision of QoS guarantees. Scout [19] seeks to associate resources with data paths rather than with users or applications.

6 Conclusions and Future Work

This paper has presented the design for RCANE, a Resource Controlled Active Network Environment, and its implementation over the Nemesis Operating System. RCANE supports the execution and accounting of untrusted code written in a safe language. Direct interference between principals is prevented through the use of a safe language. QoS interference is prevented through scheduling and accounting. Experiments showed that principals running on RCANE do experience isolation with respect to CPU time, network bandwidth and GC activity. Areas for future work include: a more developed charging and accounting model, resource control and transfer on a network-wide scale, and allowing principals more flexibility in specifying scheduling and memory usage policies.

Acknowledgements

The author wishes to thank Jonathan Smith at the University of Pennsylvania, where part of this work was carried out, and Jonathan Moore and Michael Hicks for developing the PLAN infrastructure.

References

1. D. Scott Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, September 1998. 26
2. Xavier Leroy. *Objective Caml*. INRIA. http://caml.inria.fr/ocaml/. 26, 31, 34
3. Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accountable Execution of Untrusted Programs. In *Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII)*, March 1999. 26
4. I. M. Leslie et al. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. 31, 35
5. Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1998. 31, 35
6. David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *1st IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, April 1998. 31, 34
7. C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, February 1973. 31
8. Timothy Roscoe. The Structure of a Multi-Service Operating System. Technical Report 376, University of Cambridge Computer Laboratory, August 1995. 31
9. Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol Implementation in a Vertically Structured Operating System. In *22nd IEEE Conference on Local Computer Networks (LCN)*, 1997. 32
10. Sean Rooney. Connection Closures: Adding application-defined behaviour to network connections. *Computer Communications Review*, April 1997. 34
11. D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication*, September 1997. 34
12. Michael Hicks, Jonathan Moore, D. Scott Alexander, Carl Gunter, and Scott Nettles. PLANet: An Active Internetwork. In *IEEE INFOCOM '99*, 1999. 34
13. Beverley Schwartz, Alden Jackson, Timothy Strayer, Wenyi Zhou, Dennis Rockwell, and Craig Partridge. Smart Packets for Active Networks. In *2nd IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, 1999. 35
14. Active Networks NodeOS Working Group. NodeOS Interface Specification. Draft. 35
15. Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1998. 35
16. C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *1998 USENIX Annual Technical Conference*, June 1998. 35
17. Philippe Bernadat, Dan Lambright, and Franco Travostino. Towards a Resource-safe Java. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications (PLRTIA)*, December 1998. 35

18. Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an Operating System Architecture for Application-level Resource Management. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, 1995. 35
19. A. Montz, D. Mosberger, S.W. O'Malley, L. Peterson, and T. Proebsting. Scout: A Communications-Oriented Operating System. Technical report, Department of Computer Science, University of Arizona, June 1994. 35