Technical Report

UCAM-CL-TR-561
ISSN 1476-2986

Number 561

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Resource control of untrusted code in an open network environment

Paul B. Menage

March 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted June 2000 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Magdalene College.

Some figures in this document are best viewed in colour. If you received a black-and-white copy, please consult the online version if necessary.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Summary

Current research into Active Networks, Open Signalling and other forms of mobile code have made use of the ability to execute user-supplied code at locations within the network infrastructure, in order to avoid the inherent latency associated with wide area networks or to avoid sending excessive amounts of data across bottleneck links or nodes. Existing research has addressed the design and evaluation of programming environments, and testbeds have been implemented on traditional operating systems. Such work has deferred issues regarding resource control; this has been reasonable, since this research has been conducted in a closed environment.

In an open environment, which is required for widespread deployment of such technologies, the code supplied to the network nodes may not be from a trusted source. Thus, it cannot be assumed that such code will behave non-maliciously, nor that it will avoid consuming more than its fair share of the available system resources.

The computing resources consumed by end-users on programmable nodes within a network are not free, and must ultimately be paid for in some way. Programmable networks allow users substantially greater complexity in the way that they may consume network resources. This dissertation argues that, due to this complexity, it is essential to be able control and account for the resources used by untrusted user-supplied code if such technology is to be deployed effectively in a wide-area open environment.

The Resource Controlled Active Node Environment (RCANE) is presented to facilitate the control of untrusted code. RCANE supports the allocation, scheduling and accounting of the resources available on a node, including CPU and network I/O scheduling, memory allocation, and garbage collection overhead.

A prototype implementation of RCANE over the Nemesis Operating System is described; an experimental evaluation is undertaken to demonstrate the value of such an approach. Sample implementations of existing active network systems that have been adapted to use RCANE's resource control interfaces are presented.

Acknowledgements

I would like to thank my supervisor, Simon Crosby, for his enthusiastic yet unstifling advice and encouragement during my time at the University of Cambridge Computer Laboratory. Many Systems Research Group members, past and present, have provided helpful discussions and infrastructure without which this work would not have been possible. In particular, thanks are due to Neil Stratford, Tim Harris, Richard Mortier, Dickon Reed, Steven Hand, Andrew Moore, Austin Donnelly, Stephen Early, Ian Pratt, Paul Barham and Richard Black.

For reading and commenting on drafts of this dissertation I am indebted to Simon Crosby, Steven Hand, Dickon Reed, Ian Leslie, Richard Mortier, Tim Harris, Austin Donnelly, Steven McKellar and Rebecca Isaacs. For her 2am colour printing heroics, Melissa Ho deserves to be canonised. Thanks also go to the Computer Laboratory system administrators for their tireless work.

I would like to thank Jonathan Smith for his substantial encouragement and advice, and for the very rewarding summer that I spent in the Distributed Systems Laboratory at the University of Pennsylvania. Scott Alexander, Angelos Keromytis, Marianne Shaw, Mike Hicks and Jonathan Moore also helped make my time there enjoyable and worthwhile.

This work was supported by a grant from the EPSRC. I am also very grateful to the Computer Laboratory for supporting me for an additional term beyond the end of my EPSRC grant.

Finally, I would like to thank all those friends who made my six years in Cambridge so enriching, memorable and fulfilling – this dissertation is dedicated to them, and in particular to Clare, Sean and Paul.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation argues that in order to realise practical programmable networks, robust resource reservations and policing are necessary on the programmable nodes within the network. It presents the design, implementation and evaluation of an architecture – RCANE – for the resource-controlled execution of untrusted code in an open programmable network. The architecture supports secure Quality of Service (QoS) provision to individual clients; it prevents malicious, greedy or erroneous clients from interfering with the system or with other clients by consuming excessive system resources and permits effective accounting and billing to the remote clients.

1.1 Motivation

1.1.1 The Case for Mobility

The speeds achieved by computers and communications networks are continuing to improve over time. However, in the field of distributed processing, two obstacles inevitably stand in the way of such progress. These two obstacles are the inherent latency associated with widely-distributed communications and the existence of bottlenecks within networks.

1.1.1.1 Inherent Latency

Firstly, the speed of light provides a fundamental limit on the speed of interactions between widely-separated computers. The round-trip time for interacting with a server halfway around the circumference of the earth is, at an absolute minimum, about 200ms. This assumes that the transmission proceeds at the speed of light in optical fibre[1] – transmissions in the current Internet suffer delays from switching and routing at multiple points along the transmission path, with the result that packets typically take at least twice the theoretical

[1]The speed of light in fibre is approximately 66% of the speed of light in a vacuum

minimum to reach their destination [Cheshire96]. Even techniques to adapt the Internet to use optical switching techniques can only approach the fundamental lower bound.

When it is necessary to transfer a large quantity of streamed media or other data between two widely separated locations, such latency may be difficult to avoid; however by anticipating future requests and caching data at intermediate points in the network, it may be possible to minimise the latency experienced.

Such caching and prefetching is not possible for applications and services for which each communication between two remotely located endpoints (typically a client and server) is unique. When a client is communicating with a server on the other side of the planet, and needs to make a series of inter-related queries (e.g. navigating through a web application or talking to a remote mail daemon), there are three factors affecting the time taken to complete the interaction:

- the time taken at the server to process each query,
- the time taken at the client between successive queries, and
- the transmission latency in the round-trip between the client and the server.

For a large class of interactions, and assuming a properly engineered and sufficiently well-provisioned server, the processing time at the server may be expected to be small.

If the activity is human driven (as in the case of many web application), the response time of the human is likely to be the dominating factor; furthermore, since the processing of the responses is being performed in an unpredictable and unspecifiable way – i.e. by a human brain – such processing *must* take place at the client-side. However, when the client-side processing may be computer-driven, as in the case of an email exchange, or in the case of defining a macro for automating repetitive actions in a web application, the time taken for client-side processing will be small. In this case, the high cost of the round-trip latency is the dominant factor in the time for a task to be completed.

1.1.1.2 Bottlenecks

The second obstacle is that there will inevitably be regions of the network – typically at or near the edges – where the available bandwidth is orders of magnitude lower than that available within the core of the network. This is currently the case with the vast majority of home subscribers to the Internet, who typically have 56Kbps connections over phone lines. The adoption of technologies such as cable modems and Digital Subscriber Link (DSL) is gradually improving this situation – however, it seems likely that these connections will remain bottlenecks as core network speeds increase. Further bottlenecks are possible at congested points within the network.

When attempting to process large sources of data (such as a video stream or a remote database), the time taken to transfer the volume of data over the bottleneck link – limited by the bottleneck bandwidth, rather than the latency – may be the dominant factor in the time taken. In the case of a database query, the size of the eventual results required after processing the data may be orders of magnitude smaller than the data itself. In the case of a video stream, the user may be prepared to settle for a lower quality (and hence lower bandwidth) of video if it allows the video to be received in real-time; however, if the provider of the video does not make such a low-bandwidth stream available, the user has no alternative but to transfer the entire stream over the bottleneck link. In both of these situations, the ability to filter or process the high-bandwidth stream of data to produce a lower-bandwidth representation *before* it has to cross a low-bandwidth or high-latency link would reduce the time spent transferring data across the bottleneck.

1.1.2 Programmable Networks

With the rise of the World-Wide Web, the introduction of Java [Gosling95b] applets is an example of a solution to the problem of inherent latency – by downloading program logic to the client, the server can interact with the user's display without the high latencies that would otherwise occur. However, this solution is less effective in situations where the interaction is characterised by a client querying a server to make use of a body of data maintained by that server, such as when querying a database or posting to a news server.

An alternative is to permit such computation to be moved onto or toward the server. By permitting clients to upload their own agents to be executed on the server, latency delays can theoretically be reduced to a single round-trip time, even when multiple interactions are required between the server and the client's agent. A disadvantage of moving code to the server is that it burdens the server provider with issues of security and resource control on top of any work required to implement and maintain the server – hence in many cases it may be more appropriate to move the client's agent to specialised "computation servers" close to the desired server. A further refinement, as espoused by the active networking community, is to permit such processing to be performed within the forwarding path of the network switches. This allows users to exploit application-specific networking requirements, so as to make more efficient use of the network. These two approaches of moving program logic from the client towards the server fall under the general field of *programmable networks* – permitting users of the network to utilise the resources of nodes within the network in ways not forseen by the network providers.

For the purposes of this dissertation, the term *programmable network* is used to refer to a network in which *some* of the nodes provide programmable extensions to end users. To obtain benefits from programmable networks it is not necessary that all nodes within the network are programmable; indeed, a totally programmable network would be likely to be counter-productive on efficiency grounds. The bulk of traffic in a network can be adequately served by a traditional store-and-forward service model. The latency and processing overheads experienced by flows through a programmable platform are likely to be substantially greater than those experienced through a hardware router or switch, thus forcing all traffic through a programmable platform would degrade the overall utility of the network.

The term *open* programmable network is used to imply that these programmable facilities are available to *all* users of the network (possibly in return for an additional payment), not just those users employed by or trusted by the network providers. Thus the code being executed on the network nodes may be *untrusted.* It is therefore vital that the network providers are able to prevent user-supplied code from compromising the safety of the network nodes or the stability of the network itself.

1.1.3 Untrusted Code

For the purposes of this dissertation, *untrusted code* is considered to be any executable code that is supplied by a client to some kind of server (possibly an OS kernel, or a server within the network), and that is executed by that server, but where there are no particular goals or trust shared in common between the client and the server.

Thus this definition excludes certain instances of mobile code:

- When both the client supplying the code and the server running the code share some common goals (e.g. within a single organisation, or a widespread group of co-operating researchers), it is reasonable to assume that the likelihood of denial of service (DoS)[2] attacks and excessively greedy applications is reduced. The resource control techniques presented in this paper are still applicable in such environments, in order to improve robustness or help differentiate the levels of QoS required by different activities; however, the inevitable overhead incurred by robust resource control may mean that such control is considered excessive for these situations.
- When the code is supplied by the server to the client[3], it is unlikely that resources on the client machine are as scarce as those on a server; moreover, although a DoS on a client machine is likely to cause annoyance and inconvenience to the user, it is unlikely to cause major problems to an enterprise. Thus whilst some form of resource control can be useful to prevent malicious Java applets from mounting DoS attacks on web browsers, the techniques presented in this dissertation may not be necessary.

1.1.4 Issues of Resource Control

With the increased flexibility provided by programmable networks comes a greater complexity in the ways that server and network resources – including CPU time, memory and bandwidth – may be consumed by end-users. In a traditional network, the resources consumed by an end-user at a given network node may be 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. The resources consumed by a client at a server can be similarly bounded by limiting the rate at which requests from that client are accepted. Since in each

[2]A *Denial of Service* attack is one in which an attacker gains no direct benefit, but attempts to degrade the service received by other users of a node or network.

[3]In this context a server is a multi-user system that receives requests, and a client is a single-user system that makes requests.

case the code to be executed is under the control of the server/network node, there is no straightforward way for the client to circumvent such resource limits.

When the client is permitted to specify or customise the code that is executed on the server or within the network, such resource control is more difficult. Security measures may be employed to permit only those clients trusted by the node to supply code, but this defeats many of the advantages that code mobility can provide. Language restrictions and formal proofs may be used to verify that the code performs no intrinsically illegal actions, but the task of deciding in advance whether a particular piece of code will consume excessive resources is, in general, intractable. Even in the absence of specific DoS attacks, the task of allocating resources according to a specified QoS policy is complicated by a lack of knowledge about the behaviour of the user-supplied code.

Moreover, if programmable nodes and computation servers arise within the network, it is likely that commercial pressures will require some form of charging for use of the services provided. In return, users will expect some assured level of QoS. To offer such QoS guarantees, the environment in which the code executes needs to provide isolation between different clients, and fine-grained scheduling and accounting mechanisms.

1.2 Contribution

Diverse paradigms have previously been developed to allow the remote execution of code at nodes within a network. These have included projects from within the active networks and open signalling communities, and from the more general mobile code research communities.

Many of these earlier projects have focused on the basic security mechanisms required for the safe execution of untrusted code. These mechanisms have involved the use of safe languages, proof-carrying code and high-level interpreters. Negotiation for access to resources has also been studied, particularly within the mobile code communities.

However, the mechanisms for regulating such access to resources once it has been granted, to ensure that DoS attacks are preventable and to enable the provision of effective QoS guarantees to the untrusted mobile code, have not previously been sufficiently studied.

Some projects have temporarily ignored the issue, on the grounds that the suppliers of the code were trusted not to consume excessive resources or perform DoS attacks. This is a reasonable approach to take when initially developing the high-level environment in which the mobile code must run. However, to deploy such solutions in the real world is not practical.

Those projects that have considered resource control have tended to implement policing with a very coarse granularity. Two solutions typically proposed have been:

- If the processing for any packet takes longer than a given period of time, then assume that the packet is misbehaving and drop it.
- Employ a priority-based scheduler; at intervals check that threads are not exceeding their allocated time, and reduce their priority if they are.

Both of these solutions are unnaceptable if a node is attempting to support QoS guarantees for clients, since they do not act on a sufficiently fine time scale.

Moreover, previous work on programmable networks has not sufficiently addressed the issues of safely revoking and reclaiming the resources of clients who have exited or been aborted. In a traditional operating system, using hardware protection between clients, each client is typically sufficiently isolated and self-contained that revoking the client's resources is relatively straightforward. The software protection used by many programmable network node prototypes provides a more lightweight environment; however these have typically provided insufficient control over the reclamation of some of the node resources (such as memory and threads) associated with departed clients, since there is no clear notion of resource ownership at the virtual machine level.

Providing robust resource control helps to prevent direct DoS attacks on a programmable node itself. A further problem that must be considered is that of preventing DoS attacks from being launched by user code running on the node against other sites within the network. Whilst identifying likely DoS behaviour in real-time is likely to be a hard problem, the maintenance of accounting records (required for billing) will also facilitate tracing the culprit following a DoS attack launched from a programmable network node.

The thesis of this work is that the provision of platforms for the general execution of untrusted code within a network requires an architecture with support for:

- fine-grained resource partitioning, with accounting and scheduling
- lightweight communication between clients
- effective resource revocation

This dissertation gives a rationale for such resource usage. An architecture is presented, along with a prototype implementation on the Nemesis operating system. The effectiveness of the architecture is evaluated to demonstrate that it provides the above properties. In addition, implementations of two resource controlled Active Network architectures, PLANet and ANTS, have been developed over the abstractions provided by RCANE, together with an implementation of a multicast protocol that utilises the flexibility provided by programmable networks to trade memory consumption for bandwidth and latency.

1.3 Outline

The organisation of the remainder of this dissertation is as follows.

Chapter 2 provides background material relevant to this work. The development of networks, from the earlier "passive" networks to the programmable nodes offered by active networks is discussed. Relevant research in operating systems and safe languages is also reviewed.

Chapter 3 discusses the resources that need to be controlled in the presence of untrusted code, and the extent to which such control is required. It is argued that without such control, widespread deployment of programmable nodes in a network is impractical.

Chapter 4 discusses the Nemesis operating system and the features that make it a suitable platform for the resource control of untrusted code. Comparison is made with other operating systems to establish the suitability of Nemesis as a base for RCANE.

Chapter 5 introduces the architecture for RCANE, an environment permitting the safe and resource-controlled execution of untrusted code. The principles that guided its design are discussed, and the abstractions used to allow controlled access to the resources on a node in a programmable network are presented.

A prototype implementation of the RCANE architecture over the Nemesis operating system and the Objective Caml (OCaml) language is presented in Chapter 6. An overview of the implementation is followed by an examination of the modifications made to OCaml to adapt it to the requirements of the RCANE environment, and a description of important aspects of the implementation.

Chapter 7 provides an evaluation of the key features of RCANE through the use of microbenchmarks, and demonstrates that it provides an efficient mechanism for resource isolation between multiple clients.

Chapter 8 examines the use of RCANE in the context of Active Networks; two different styles of active networks and their implementation on RCANE are presented.

Finally, Chapter 9 summarises the main arguments of the dissertation and suggests avenues for future research.

Chapter 2

Background

This section provides background information to the development of the architecture presented in this dissertation.

2.1 Evolution of Network Services

2.1.1 Passive Networks

There has been much study of traditional *passive* networks – the most pervasive example being the collection of networks communicating through the use of the Internet Protocol (IP) [Postel81]. This provides a common interoperability layer throughout the Internet. Although clearly some form of processing (whether hardware or software) is occurring within the routers and switches in a passive network[1], the essential feature is that the interface presented to the user deals only in terms of *data* rather than *code* or customisability – the user has no way to extend or customise the processing on a node within the network. Furthermore, although in many modern switches and routers the *provider* of the node can upgrade the software on the node, such control generally occurs over a relatively long timescale.

The development of passive networks such as the Internet was guided by the principles given in the *end-to-end argument* [Saltzer84], which states that certain functions of a distributed system

> can be correctly and completely implemented only with the knowledge and help of the application standing at the end points of the [distributed] system

– i.e. that supplying complex functionality within the network is non-optimal, since it is probable that such functionality will either duplicate work that the application needs to carry out at a higher level, or will fail to meet the needs of most users. Following the thesis of

[1]The development of purely optical switches may herald the arrival of the truly passive network.

these arguments, IP supports only an unreliable, unsequenced form of data delivery. Higher level protocols such as TCP and UDP are implemented only at the end-points of the network[2], and IP must be mapped over all link layers.

The service abstraction presented by passive networks is that of a connectionless *store-route-and-forward* model for transporting packets from a source host to a destination through a series of routers. All routers process packets using conceptually the same algorithm, parametised only by certain header fields in the packet (primarily the destination IP address).

Packet forwarding over connectionless protocols such as IP involves a routing stage, typically based on a longest-prefix match of the destination address against a routing tree, which may be both expensive and complicated [Waldvogel97]. Providing different levels of service based upon other parameters such as source address and protocol increases the complexity further [Srinivasan98].

Wide Area Network (WAN) technologies such as X.25 [Jacobsen80], Asynchronous Transfer Mode (ATM) [Fraser93] and Multiprotocol Label Switching (MPLS) [Callon97] have attempted to make use of simpler connection-oriented forwarding models. These aim to reduce the cost of packet forwarding by performing the routing stage once at connection setup time, then performing a circuit look-up (which may be substantially cheaper than the routing operation performed during IP forwarding) at packet processing time. By simplifying the packet processing operation, it is possible to increase the throughput of a switch, particularly if such simplifications result in an algorithm that lends itself to implementation in hardware.

QoS provision within passive networks has been addressed in several ways. Two of the primary solutions being developed for the Internet are:

Integrated Services allow each network flow or class of flows to be mapped to a particular QoS class [Braden94]. This QoS class is used to schedule the transmission (and limit the buffering) of packets in these flows. RSVP [Zhang93] has been proposed as a protocol for allowing the creation of flow specifications at routers.

Differentiated Services attempt to reduce the amount of state required within the core routers by requiring end nodes and intermediate routers to mark each packet with the required per-hop forwarding behaviour [Blake98]. Service Level Agreements (SLAs) may be set up between different network domains specifying the traffic profile (such as the average and peak rates, and burst size) for different classes of packets.

2.1.2 Programmable Networks

In a programmable network, the software on the network node may be customised or extended at relatively short timescales; such customisation may be over the lifetime of a connection, or even carried with each packet. This may be in order to perform packet processing (*Active Networking*), connection control (*Open Signalling*), or more general extensibility.

[2]Note that a router within the network may be considered an end-point when specifically addressed as the destination in an IP packet.

2.1.2.1 Active Networking

The Active Networking [Tennenhouse97, Campbell99] community proposes moving computation into the network in order to increase network flexibility, by replacing the simple packet forwarding model used by IP with a richer programmable model. Thus instead of allowing a user to only specify the parameters (such as destination address) used in packet processing, an active network permits enhancement or replacement of the forwarding routines themselves. The benefits that can be gained from such programmability include:

Customised routing: Customised routing algorithms could allow applications to circumvent deficiencies in the standard IP routing behaviour or to tailor the QoS properties of the route to the needs of their traffic.

Deployment of new protocols: Session/transport-level protocols such as HTTP and TCP may be deployed purely at the end hosts, with no modifications required to network routers. However, deploying new network-level protocols – such as IP multicast extensions [Deering89] – over the Internet requires long periods of time both for standardisation, and for implementation by router vendors. By providing a programmable environment in which users can define custom packet processing routines, network-level protocols can potentially be deployed substantially more quickly. Examples of such protocols deployed over active networks include reliable multicast [Lehman98] and transparent web cache redirection [Legedza98].

Access to non-standard resources: In the current Internet, router-specific resources (such as enhanced capabilities on a novel link-layer) cannot easily be utilised by end users, since there is no way in the basic IP protocol to indicate a requirement to use such resources. By specifying a more flexible, extensible and programmable network substrate, users can more easily make use of enhanced capabilities at certain nodes within a network.

Dynamic adaptability: Programmability allows a flow to be dynamically processed at certain points within a network, such as to add FEC (Forward Error Correction) at lossy points within a network [Hadzic98], or to transcode a high-bandwidth multicast stream into a lower bandwidth stream before sending it over a low-bandwidth link [Amir98].

Each of these advantages could be realised to some extent by making use of the IP *options* [Postel81] mechanism to communicate such requests for non-standard behaviour to routers and endpoints. However, for this approach to be generally useful, some specification for the required extensibility and programmability must be reached. In this respect the IP option mechanism is not ideal – although some specific options are defined, there is no accepted standard for extensibility. Prototype implementations of active network systems have been developed to encode capsule code within IP options [Wetherall96, Murphy97]; however, to be of general practical benefit, such use of options requires more uniform standardisation. Furthermore, the use of IP options typically results in a packet experiencing lower-quality processing throughout the whole of the path it traverses; packets with options set tend to be queued with lower priority, due to the extra complexity that they present to a router. The

Active Network Encapsulation Protocol (ANEP) [Alexander97a] has been proposed as a more flexible mechanism than IP options for encapsulating active network packets for transmission over different lower-level media and protocols. It supports the multiplexing of packets from multiple *execution environments* (EEs) over a single channel.

In [Bhattacharjee97], it is suggested that the end-to-end argument [Saltzer84] (discussed in Section 2.1.1) does not preclude the existence of active and programmable networks; it is argued that the fundamental rationale behind the end-to-end argument is that:

> Some services require the knowledge and help of the end-system-resident application or user to implement, and so *cannot* be implemented entirely within the network.

and that the corollary of this premise is that:

> Some services may best be supported or enhanced using information[3] that is only available inside the network.

Thus the addition of programmability allows end users to implement precisely the services that they need – by executing user-supplied code within the network, the application-specific logic required by the end-to-end argument may be combined with additional knowledge regarding the state of the network and with the higher bandwidth and lower latency afforded to nodes within the network. In [Saltzer98], the originators of the end-to-end argument agree with this suggestion in principle, but caution that end-to-end arguments should still be considered on a case-by-case basis to determine whether such programmability is beneficial.

2.1.2.2 Open Signalling

Analagously to the drive provided by Active Networks to open up the routing and packet processing functions on the data path of a network node, research into *Open Signalling* has attempted to provide a programmable interface to the control plane of a network node, typically a router or an ATM or telephony switch.

The control plane of a communications network is responsible for setting up, maintaining and tearing down connections between switches within the network. The interactions required for such activity, both between entities within the network and between the network and end hosts, is known as signalling.

Historically, telephony networks existed to carry voice connections between end-points. The services available were defined by the network providers, and updated over long timescales. The only signalling available to users of the network was that for setting up point-to-point calls (i.e. dialling). Within the networks, out-of-band signalling mechanisms such as Signalling

[3]The author feels that this sentence would have been more effectively phrased "...information *and resources* ..."

System 7 [Jabbari91] were developed to provide greater security and scalability. Intelligent Networks (IN) services were developed over SS7, permitting the deployment of more advanced (network-provided) services such as conference calls and toll-free access numbers. Provision of new and more sophisticated services by end-users or third parties was not possible.

Initial attempts to define standards for broadband (ATM and ISDN) signalling [ATMF96, ITU-T94] followed paradigms inherited from the telephony standards and thus tended to limit users to only those services provided by the network operator.

Alternative signalling strategies – known as *control architectures* (CAs) – such as [Crosby95, Hjalmtysson99, Newman96, Lazar96] have been proposed; each with its own advantages and disadvantages. In [van der Merwe97] it was argued that no single control architecture could provide for the needs of all applications and services.

Projects such as the Tempest [van der Merwe97, Rooney98b] architecture from the University of Cambridge aim to provide an environment to support flexible management of connection-oriented networks. Within the Tempest, the physical resources at an ATM switch are partitioned through the use of a switch divider into multiple logical *switchlets* [van der Merwe96, van der Merwe97]. Each switchlet appears to its users as an independent switch with its own range of virtual paths and virtual circuits. Multiple CAs may then be run on the network, each seeing the appearance of its own private network of ATM switches. Each CA has full control over the range of VCIs and VPIs assigned to its switchlet, allowing it to manage the connections established within the virtual network. When a virtual network is created, a general CA such as PNNI [ATMF96] or IP Switching [Newman96] may be employed, or alternatively a *Service Specific Control Architecture* may be instantiated to provide better use of network resources for a particular application ([van der Merwe97] presents the example of videoconferencing). Moreover, components of the architecture (such as a switch divider or a CA) may be extended through user-supplied code to provide custom processing within the network.

2.1.2.3 Level of Programmability

Research into active networking and open signalling has been undertaken at all levels, from the physical layer [Hadzic98, Lee99] to the application layer [Amir98, Fry98]. Furthermore, the level of programmability provided by different projects has varied considerably.

Capsules: At the most fundamental level of programmability, each packet contains both executable code and data. Such a packet is often referred to as a *capsule*[4]. When a capsule arrives at a router, the code carried by that capsule is executed by the router, typically parameterised by the capsule's data (*payload*). Possible actions performed by the code include:

- Performing a routing decision and transmitting itself (or multiple copies of itself) out along egress links from the router.

[4]Note that the terminology in the field of Active Networks is not yet well-defined.

- Creating fresh capsules, and sending them on to new destinations or back to the original capsule's source.

- Interacting with objects or other state in the router, either to obtain information about the router or local network, or to perform some control operation.

Alternatively, each capsule may contain mostly data, but with some specification of the code that should be used to process the capsule[5]. If the specified code is not found, the capsule may be dropped, or the code may be obtained via some out-of-band mechanism.

The ALIEN [Alexander98a] architecture from the University of Pennsylvania and the SmartPackets [Kulkarni98] project from Kansas University both support capsules. An ALIEN capsule consists of an OCaml [Leroy97] bytecode module and a payload; a SmartPacket contains the serialised class definition and state for a Java object.

The Packet Language for Active Networks (PLAN) [Hicks98, Hicks99c] is built around the two concepts of invoking named services on a node, and invoking a function on a remote node – such a remote invocation is semantically (and functionally) equivalent to sending a packet. A PLAN capsule contains a *chunk*, which represents a delayed remote PLAN evaluation. (See Section 2.2.2 for more details on PLAN chunks.)

The Active Node Transfer System (ANTS) [Wetherall98] and PAN [Nygren99] from MIT support demand-loading and caching of code at a network node. Each active capsule is tagged with a code identifier. The code identifier is implemented as a secure hash (such as MD5 [Rivest92] or SHA-1 [NIST95]) of the Java [Gosling95b] class (ANTS) or generic code object (PAN) that should process that capsule. This prevents malicious users from hijacking other users' capsules, since the only way to receive another user's capsules is to preload exactly the class/code that the originating user intended to be used for processing the capsule. The payload of the capsule consists of a serialised representation of an instance of the specified class (ANTS) or generic data (PAN). In the event that the required code is not available at the receiving node, a request is sent to the upstream neighbour (ANTS) or the specified code repository (PAN) from which the capsule was received, to obtain the class/code for processing that capsule (and any associated capsule types in the same protocol). In this way, a capsule pulls the code required to process it through the network. Since code caching is used, subsequent instances of the same capsule type (within a period of time determined by the level of caching) will be serviced immediately, without the delay of requesting code from elsewhere in the network.

Capsules may be transient (as in the case of ALIEN) or they may be permitted to maintain persistent state at network nodes. The PLAN environment provides a `resident` service that permits related packets to share persistent state.

The concept of capsules is also applicable within the area of open signalling. Active Reservation Protocols [Braden99] allow Java capsules to be used for signalling (passive) network flow requirements to a QoS enabled router, thus permitting greater flexibility when specifying

[5]Such a specification is, of course, semantically very similar to the headers on an IP packet, but with enhanced flexibility.

flow classes. The Smart Packets[6] [Schwartz99] project from BBN Technologies aims to improve the flexibility and reduce the bandwidth and latency required for network management. Capsules carrying management programs may travel around a network from device to device interrogating and modifying SNMP [Case90] MIB databases, only needing to send packets back to a monitoring station in the case of an abnormal situation.

The *Caliban* [Rooney98a] interface within the Tempest permits similar SNMP querying through the use of Java bytecode transported across a network. The *Elastic Network Control* provided by the *Haboob* architecture [Bos99] supports remote invocation using *granules* (autonomous program units) to customise behaviour throughout all levels of a network.

Extensions: At an intermediate level of programmability, *active extensions* may be loaded on to a router or switch by an out-of-band mechanism. Associated with each extension is a set of flows, whose packets should be processed by this extension.

ALIEN and PLAN both support the loading of OCaml extensions; this facility was used to create an extensible active Ethernet bridge [Alexander97b] running over ALIEN. Extensions register packet filters to give access to the network flows that they wished to process. The Composable Active Network Elements (CANES) [Bhattacharjee98] architecture supports a limited form of programmable extensions – the behaviour of a flow of packets may be customised by selecting *triggers* that are called by pre-defined programs at certain points during packet processing.

Network hardware may also be extended: the Programmable Protocol Processing Pipeline (P4) [Hadzic98] and other projects such as [Lee99] allow users to download processing logic into field-programmable gate arrays (FPGAs), permitting high-speed custom processing at the link layer and physical layer.

A generic extension intended to process a large class of packets may be classified as an *execution environment* (EE). An EE may itself permit extensibility by the end users of the network, either by directly executing the packets that it is processing, or through out-of-band mechanisms. Systems such as ANTS and PLAN may themselves be implemented as EEs over a lower-level *NodeOS* (see Section 2.1.2.4), allowing multiple active network environments to co-exist on a single node.

Partially Programmable Networks: Not all proposals for active networking require that the entire network be made programmable. [Smith98] presents *Active Router Control*, a hybrid active/passive architecture which assumes a passive IP forwarding network as the main "transport plane", and associating a "controller" with groups of IP routers, to deal with higher level issues such as maintenance of routing tables. The controller itself may be an active node, thereby permitting flexible customisation of the network by users, without degrading the performance of standard IP flows that have no requirement for active behaviour.

A similar approach to partially programmable networks has resulted in the development of the Active Networks Overlay Network (ANON) [Tschudin99]. ANON supports the creation

[6]Not to be confused with the SmartPackets project from Kansas University.

of programmable clusters of active nodes – called *segments* – connected together by a larger overlay network.

Active Services: At the highest (and least flexible) level of customisation, Active Services [Amir98] (also referred to as *Application Layer Active Networking* [Fry98]) permit programmability within the network, but only at the application layer. User-supplied code may be executed on nodes within the network, but cannot replace the processing used for the network or transport layer. This allows users to experience reduced latency by moving computations closer to the nodes with which they are interacting (e.g. a web document could specify code to be executed on a programmable web cache, giving faster responses than by connecting through to the original web server on each request [Marshall99]).

Xenoservers were proposed in [Reed99] to provide hosts at strategic points within the network that can supply execution services to untrusted clients, in return for payment. In many respects the concept of Xenoservers is similar to Active Services but aims to present an environment in which resource usage is strictly scheduled, accounted and charged for.

Within the open signalling community, projects such as the Hollowman [Rooney97] Tempest CA support extensible control of ATM switches. Users of these architectures may supply code in a safe language – Java in the prototype Hollowman implementation – to execute on or near a switch in order to deploy application-specific policy for a particular connection. Such code can interact with the virtual network's switchlets, to configure its VCI mappings and resource allocations with lower latency than code executing at end points of the network. This enables more effective control over the resources associated with the CA's virtual network.

The *Haboob* [Bos99] takes the extensibility of Hollowman a stage further with the concept of *Elastic Network Control.* It proposes *Sandboxes* in which Dynamically Loadable Agents (DLAs) may be executed. Sandboxes allow DLAs to present extensible interfaces that may be externally invoked – either by other DLAs or by external applications – by using the Simple Uniform Framework for Interaction (*SUFI*). Sandboxes may be instantiated at any location with the network control and management architecture where extensibility is desirable. In the example implementation, sandboxes are presented for the execution of DLAs written in Java and TCL [Ousterhout90].

2.1.2.4 Platforms

The environment over which active network execution environments and open signalling and mobile code systems are implemented has a fundamental effect on the properties of such systems. Previously, the majority of active network projects have been developed over various flavours of the UNIX operating system. In particular, open source variants such as Linux have been widely used due to the ease with which kernel modifications may be made, facilitating modifications to network packet processing.

Running an active network system in a Unix user-space environment has the advantage of being easily portable among different POSIX-compliant [NIST88] operating systems. However, the abstraction presented by the operating system often obscures a great deal of the detailed information and control over the data-path that a low-level active network system requires to perform activities such as resource scheduling.

Several projects have attempted to move all or part of the network programmability inside the OS kernel. The Network Element for Programmable Packet Injection (NEPPI) [Cohen99] is a Linux IP router extended to allow programmable translations on certain TCP flows. *Gateway programs* may request that packets on flows which match a specified filter are passed up to the user-level processing routine, which may manipulate the packet directly, or may program a specific manipulation (such as packet redirection, address translation or adjustment of TCP sequence number and window size) into the router's forwarding tables for future packets on that flow. By specifying TCP flag bits as part of the filter, a gateway program may for example receive all connection setup packets (with the `SYN` flag set) on a particular flow, but allow all established connections to be processed within the kernel router. Router *plugins* [Decasper98] and the Click router toolkit [Morris99] have been proposed to support a similar – but lower level – functionality in an extensible router, with modules implemented for various functions that a router may be required to perform, and connections set up between modules and the underlying platform.

PAN [Nygren99] supports an active network implementation running unsafe native code entirely within a Linux kernel loadable module; it aims to perform zero-copy active routing for those packets that are simply being custom-routed and forwarded to their ultimate destinations.

Other projects have recognised that the I/O model of a traditional Unix workstation is not optimal for supporting extensible network programmability. Previous research has addressed both the software and hardware architecture requirements.

The Lancaster Active Router Architecture (LARA) [Cardoe99] provides a hardware/software hybrid solution for implementing active network nodes at the edge of moderate sized networks. LARA uses a dedicated processor to act as a forwarding engine for each network port, and a management node to oversee the system. Data-path communication between the forwarding engines is performed over a high-bandwidth interconnect – the prototype uses a 0.5Gb/s modified SCSI bus. Control-path communications, including interactions with the management node, use a standard bus. LARA/PAL (Platform Abstraction Layer) provides a platform-independent interface over which different EEs may be instantiated.

The "holy grail" of active networking research is the Node Operating System (NodeOS). The NodeOS [Peterson00a] interface is defined as part of the Active Networks Working Group Architectural Framework [Calvert98]. It is intended to provide a minimal common fixed point for active networking in the same way that IP provides a common fixed point for passive Internetworking.

The NodeOS is, as its name suggests, an OS-like interface over which multiple execution environments may operate. It uses the abstraction of a *flow* to represent the resources associated with remote principals, and to manage resource accounting, scheduling and admission control.

Each EE defines a networking environment tailored for some specific (or generic) requirement, and presents that environment to a subset of end-users of the network. The environment

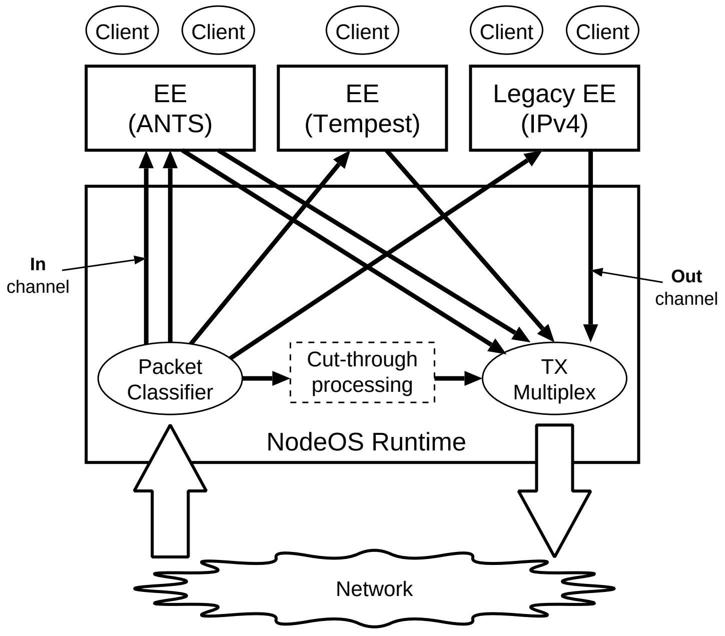


Figure 2.1: Structure of the NodeOS

provided by the EE does not have to be programmable in itself – the EE may choose to utilise the abstractions provided by the NodeOS to provide some form of customised passive packet forwarding. In fact, IP itself may be regarded as a “legacy” EE, which sits alongside other EEs and provides simple best-effort forwarding according to the node’s default Internet routing tables.

The NodeOS allows flows to open *channels* on to the network. Such channels come in three flavours:

- *In channels* allow a flow to receive packets matching some demultiplexing specification.
- *Out channels* allow a flow to transmit packets.
- *Cut-through channels* allow a flow to join an In channel to an Out channel via a specified standard routing/processing algorithm. The routing and processing of packets in a cut-through channel is performed entirely within the NodeOS, and so may potentially be optimised.

Figure 2.1 illustrates the architecture supported by the NodeOS. Currently, the NodeOS specification is at a draft stage. Several projects [Lepreau99, Hartman98, Merugu00] are developing implementations that are intended to guide the evolution of, and eventually conform to, the final specification.

2.2 Safe Execution of Untrusted Code

2.2.1 Overview

Some active networks projects have produced environments that execute arbitrary native code. PAN [Nygren99] provides such support in order to measure the inherent overhead associated with various levels of activation within the network – execution of safe Java code is also supported. Bowman [Merugu00] executes only unsafe code; the authors state that

> The choice of [the] C programming language to implement the Bowman system call interface has given us good performance, but probably sacrifices some flexibility (code can only come from *a priori*-trusted parties).

This is something of an understatement – one of the fundamental advantages that programmable networks offer is the ability for network providers to offer flexible and extensible access to their switches and routers, upon which users and third-parties can develop advanced services. Restricting such service provision to just those who are fully trusted by the network provider defeats the purpose of an open programmable network.

Furthermore, debugging applications when portions of the application's logic and data are spread throughout disparate nodes in the network is a substantial challenge. The use of some form of safe language for network programability substantially decreases the chances of uncaught bugs, and increases the probability that sensible diagnostics may be obtained in the event of a runtime error. Trusting another party not to load malicious native code on to your node is insufficient in a programmable network – you also have to assume that they are capable of writing code without the possibility of arbitrary memory corruption, due to behaviour such as running off the end of a buffer or using a pointer to previously freed storage.

Therefore, many programmable networks and mobile code systems enforce some form of safety requirements on the user-provided code that they execute. This may be implemented either by requiring use of a safe language, or by permitting code to be written in an arbitrary source language while requiring that the executable code conforms to certain well-defined properties.

2.2.2 Language-based control

Many mobile code and programmable network systems specify that the end-user's code must be written in a particular safe language. [Alexander98a] and [Cugola97] analyse the requirements for a language for mobility and network programmability.

Some systems reuse an existing safe language. Java [Gosling95b] has rapidly gained great popularity as a language for mobile code, due to its portability, type-safety and security[7]. Java is

[7]Due to incompatabilities between implementations, and to errors in bytecode verifiers and security policies, these features have been somewhat compromised. However, it is plausible that these problems will be resolved in the future.

a strongly object-oriented language with hybrid static/dynamic typing. ANTS [Wetherall98], PAN [Nygren99], SmartPackets [Kulkarni98], Hollowman [Rooney98a] and the *Haboob* [Bos99] make use of Java in slightly different ways – ANTS and PAN process capsules using demand-loaded Java classes; SmartPackets capsules contain the full definition for a Java class as well as the serialised data for a single instance; Hollowman and *Haboob* support the dynamic loading of Java classes to perform control-path operations on network switches and control architectures. Caml [Leroy97] is a dialect of ML [Milner97] that has been used as a mobile code language by RCANE [Menage99] and several components of the Switchware [Alexander98b][8] project including ALIEN [Alexander98a] and the Secure Active Network Environment (SANE) [Alexander98c]. The TCL [Ousterhout90] scripting language has also proved to be a popular base for mobile code; the *Haboob* supports mobile agents written in TCL, and Safe-TCL [Borenstein94], Agent TCL [Kotz97] and TACOMA [Johansen95] all extend TCL, with support for migration and confinement. The untyped nature of TCL, in which all values are represented (conceptually) as strings – procedures interpret these strings as being of the appropriate logical type – allows for simple but unstructured interfaces between various components in a mobile code system. The TUBE [Halls97] environment utilises properties of the Scheme language to support the suspension of mobile agents to package up the state of their computation; this state may then be transferred to a different site and the computation continued.

Other projects have developed their own languages that are specifically designed to have suitable properties for mobile computing. [Schwartz99] defines a safe high-level language, *Sprocket*, its mapping into a restricted CISC "assembly language", *Spanner*, and a virtual machine environment in which Spanner programs (*Smart Packets*) can be executed. An important feature of the Spanner environment is the addition of language level (single pseudo-opcodes) support for interrogating and modifying SNMP [Case90] MIB databases. This tight integration with a particular network management architecture reduces the system's flexibility; however, it helps fullfil one of Spanner's design goals, that meaningful programs should be expressable in a single unfragmented packet, typically under 1KB. Inferno [Lucent96] was developed to provide an operating system for distributed services such as telephony switches and media servers. It supports the execution of programs in a high-level type-safe language, *Limbo*.

The Switchware project has developed the Packet Language for Active Networks [Hicks99c]. PLAN is a restricted functional language similar to ML [Milner97], but with strong dynamic typing. The language restrictions allow PLAN programs to be safely executed without verification or checking. Remote invocation and layered encapsulation are supported through the notion of a chunk, a first-class object representing a suspended PLAN function call [Moore99b]. When a chunk is created, a string is marshalled consisting of the name of the function or service to be invoked, the arguments to the function, and the definitions of any PLAN functions required for the execution. Function names are not bound to actual functions until execution time, allowing a PLAN program to refer to services that exist on remote nodes but which are not resolvable at compilation. The `OnRemote()` primitive causes a chunk to be executed on the specified remote host. Once created, chunks may be manipu-

[8]In [Moore99a] and [Hicks99a], two of the Switchware developers discuss their experiences with an earlier version of some switchware components written over Java, and comment on the advantages that they gained after converting to use Caml.

lated by a PLAN program or the services that it invokes. For example, fragmentation may be performed by splitting the binary contents of a large chunk into several portions, and using each portion as one of the arguments to a chunk that invokes the `reassemble()` service on a remote node. Encryption, compression, reliable transfer and other network services may be performed in similar ways.

[Wakeman98] discusses various requirements for a language for programmable networks, and proposes *Safetynet* as a new language that supports these requirements. Safetynet aims to encode properties such as security and resource control into the language. This allows more substantial type checking beyond simple safety properties.

Some languages such as PLAN support dynamic scoping of identifiers, such that the unresolved bindings (for services and data) required by a mobile program must be satisfied at the executing site. Other languages such as Obliq [Cardelli94] use lexical scoping, whereby all free bindings must be resolvable at the point when a program fragment is transmitted over the network, and use of those bindings will effect remote invocations on objects at the originating site. Such lexical scoping enables a program fragment to see a more consistent view of the state of the world; however, this high level of network transparency causes both unpredictable performance and difficulties in specifically referring to state and services on the execution node. In this respect, language primitives such as PLAN's `OnRemote()` combine convenient access to remote invocations with the clear distinction between local and remote accesses.

2.2.3 Low-level control

Requiring the mobile code that a server executes to be written in a safe language has the disadvantage of reducing the flexibility available to remote users. An alternative approach taken by some researchers has been to develop ways to permit arbitrary machine code or assembly language to be executed safely.

Software Fault Isolation (SFI) [Wahbe93] runs each untrusted code module in a separate *fault domain*. A fault domain is a region of memory associated with the untrusted module, within which all loads and stores – and most jumps – are unrestricted. Code to be run is required to execute special bounds-checking code before any memory access, to ensure that the address being accessed is within the permitted data area[9], or in the case of a jump, that the address being jumped to corresponds either to a valid basic block or to an entry point in a trusted *jump table* to support cross-domain RPC. This access discipline may be enforced in two ways. In the first instance the end-user may employ a specially modified compiler that inserts the relevant checking code, combined with a verifier that runs on the server to ensure that the checks are present. Alternatively, the server may use binary rewriting techniques to insert the necessary checks before loads, stores and jumps when the code is first supplied. This allows any compiler to be used on the client[10].

[9]Extensions to the SFI model have permitted multiple data segments to be accessible to a single fault domain; these segments may be shared between multiple domains.

[10]Current implementations of SFI do place limitations upon the register usage of the code being isolated, since some registers are required to be reserved for efficient fault isolation.

The encapsulation provided by SFI allows any untrusted code to be safely executed; however, this is at the expense of an execution time overhead of approximately 5%, and with a relatively crude protection model. Other research has focused on the use of formal methods to reduce the overhead of untrusted execution while providing a richer typing model. *Typed Assembly Language* (TAL) [Morrisett98] and *Proof-Carrying Code* (PCC) [Necula97] have both been developed as ways of providing a typing discipline for low-level code. TAL supports a generic type system for basic blocks, over the registers and stack of a processor, with higher-order extensions to cope with the low-level control possibilities available to assembly code. For example, part of the typing of a basic block might specify that the `RA` (Return Address) register must contain a pointer to a block which expects register `R0` to contain an integer; thus any piece of code that causes a control transfer to the basic block must ensure `RA` contains a pointer to an appropriate return address or continuation. TAL thus ensures that the typing discipline is respected without prohibiting the calling conventions of any particular language (such as tail-call recursion or continuation passing). This work has been extended [Hornof99] to allow the partial generation of certified code, which may be verified in a similar manner to Java; the final compilation stage proceeds on the execution host, taking advantage of properties of that system. PCC allows the executing host to publish arbitrary sets of pre- and post-conditions for code segments that must be respected by a client's code. These conditions may determine both the behaviour of code supplied by the client, and the arguments that may be passed to callback routines supplied by the host. From these conditions and from analysis of the code, a set of *verification conditions* may be generated and proved offline by the client, through the use of a theorem prover. The client presents this proof to the host along with the code; verifying that the proof is valid is significantly less expensive than generating the proof initially.

2.2.4 Security

Safety is concerned with enforcing the low-level characteristics of an environment for the execution of untrusted code. Security, on the other hand, involves discovering what resources a program is authorised to access, and ensuring that a program does not gain unauthorised access to resources or information.

Whilst a sensible security policy is recognised as vital for the effective deployment of a programmable network node, the development of such a policy is largely beyond the scope of this dissertation. This section therefore briefly reviews a selection of security solutions proposed for open and programmable networks, rather than surveying wider research on the general issue of security.

At the level of inter-node cryptographic security, the Secure Active Network Environment (SANE) [Alexander98c] allows peers in an active network to establish trust between each other, by exchanging certificates that authenticate them, and to establish secret keys to permit secure communication. The firmware in the nodes is modified to provide a Trusted Computing Base.

Security in PLAN is achieved by restricting the namespace of services available to a PLAN program according to the capabilities that it carries [Hicks99b]. The services available through the namespace are implemented in a general-purpose programming language, thus requiring verification before installation. Under the PLAN security scheme, a firewall may be implemented by encapsulating a PLAN *chunk* (see Section 2.2.2) in a function call that severely restricts the packet's namespace and then evaluates the original chunk.

The Active Networks Security Working Group are defining the Security Architecture for Active Nets [SecArch98]. This is intended to be developed in tandem with the NodeOS to provide tailored security facilities.

Oasis [Hayton96] and KeyNote [Blaze99] both provide security policy architectures for services in an open network – in each case policies may be specified in a domain-specific language, and enforced upon the entities in the network.

2.2.5 Operating System Extensions

In traditional operating systems, the user/kernel division provides an inflexible interface; applications may use only those abstractions and services which are provided by the interface. This may result in inefficient behaviour, with substantially greater amounts of time spent in user/kernel crossings and memory copying than would be necessary if a different abstraction were being used.

Many operating systems permit the loading of *modules* into the kernel. These are generally used to provide support for particular of hardware. They may also be used to extend the abstractions offered by the kernel, by providing extension system calls or a pseudo-device. Non-administrators are generally not permitted to load their own kernel modules, even when the module does not in itself extend the privileges of the user (i.e. it only performs actions that the user would be permitted to perform, but in a more efficient manner), since there is usually no support for verifying that such extension modules are safe.

To this end, various research projects have developed extensible operating systems to permit users to employ more efficient abstractions for accessing the resources on a computer.

SPIN [Bershad95] from the University of Washington permitted extensions written in Modula-3 [Cardelli89] to be loaded into the kernel by user-processes, with the intention of improving performance by reducing the number of user/kernel crossings. User-level programs were run under hardware protection, and so could be written in any language. In the prototype, kernel extensions had to be signed by a trusted compiler.

Since users were permitted to load extensions to respond to interrupt events, SPIN extended the Modula-3 language with an `EPHEMERAL` keyword; `EPHEMERAL` procedures could only call other `EPHEMERAL` procedures, and hence could not lock critical resources belonging to the kernel (which were only accessible through non-`EPHEMERAL` procedures). Thus they could be safely aborted by the kernel if they spent too long in an interrupt service routine.

VINO [Seltzer96] solves the problem of resource-hoarding by kernel extensions in a different way. The Vino kernel is extensible through the use of SFI (see Section 2.2.3). The extensions are all invoked through a transaction system, which allows the kernel to abort extensions and undo changes made to system state by the tardy extension.

The Exokernel [Engler95, Kaashoek97] project from MIT supports the notion of *OS-in-a-library* in a similar way to Nemesis (see Chapter 4), although the primary motivation is that of improved performance due to reduced abstractions, rather than the goal of improved QoS enforcement that underlies Nemesis. As such, its notions of resource isolation between processes are less well-defined.

The Exokernel permits extension in several ways. *Dynamic Packet Filters* [Engler96] allow applications to pass flow demultiplexing specifications down to the network driver. *Untrusted Deterministic Functions* [Kaashoek97] (UDFs) allow library-based file systems to pass meta-data parsing routines down to the kernel, where they may be safely executed. By initially emulating the instructions of a UDF on a particular piece of meta-data, the kernel can be satisfied that the UDF will always return the same results for that meta-data, and hence can run the UDF natively on future occasions.

2.3 Resource Control and Accounting

Much research has been done into mechanisms for controlling the resources allocated to entities executing on a computer or within a network.

2.3.1 Resource Control in Operating Systems

Traditional operating systems such as Unix, as well as more recent micro-kernels, are typically optimised for performance, often at the expense of reliable resource partitioning amongst the different tasks occuring on the system.

A major factor in such unreliability has been the tendency to move processing into the kernel (in the case of monolithic systems) or into shared servers (in the case of micro-kernels). This can cause crosstalk in the timeliness of receiving access to resources, and also to obscure the amount of resources consumed by each client on the system. Whilst some OS kernels provide real-time scheduling of the CPU, such crosstalk and obsfuscation mean that simply scheduling the CPU is insufficient.

Some approaches to solving this problem have attempted to fit such resource control on to traditional systems. LinuxSRT [Ingram99] modifies the Linux scheduler to provide guaranteed levels of access to the CPU, and further demonstrates that by adding schedulers to shared servers it is possible to provide QoS in those servers, and thus avoid some of the crosstalk.

A major source of crosstalk in traditional operating systems arises due to network protocol processing, which in a monolithic kernel is often performed in interrupt handlers. In an attempt to reduce this problem, techniques such as Lazy Receiver Processing [Druschel96] and Signaled Receiver Processing [Brustoloni00] more properly account to applications the time spent performing protocol processing their incoming streams of packets. This is achieved through the use of a packet filter to associate incoming network data with a particular local socket. This data is then left unprocessed until such time as a `read()` is requested on that socket. In addition to more accurate accounting, a further benefit of this technique was observed – the improved temporal locality of the network data resulted in a lower number of cache misses. In a traditional network stack the data is pulled into the cache on receipt (for header checking, checksumming, etc.); by the time the data is copied to user-space for the receiving application, it is likely to have been displaced from the cache. By only processing the packet data directly before it is copied to user-space and used by the application, the level of cache pollution is reduced.

In [Banga99], *resource containers* are proposed as a means whereby resource accounting and protection may be separated. A resource container is associated with a particular activity and may be shared between multiple processes; similarly, a process (particularly servers) may hold multiple resource containers – corresponding to different clients or classes of client – allowing it to inform the OS to which of its containers it desires its current activity (e.g. the consumption of CPU cycles, or the protocol processing for the traffic on a particular socket) to be accounted. Eclipse [Bruno99] provides a similar form of resource control, using a *reservation file system* to model hierarchical resource guarantees.

Scout [Mosberger96b] takes the two preceding ideas a step futher, representing all activity in terms of *paths*. The informal notion of a path, commonly used in terminology such as "fast path" and "data path", is formalised to consist a set of modules – referred to as *routers* – that process a particular class of data sequentially. For example, an incoming video stream being displayed as part of a video conference would be represented by a path that begins in the network subsystem. A packet classifier in the link-level (Ethernet) device driver module would recognise packets that constitute the video stream and direct them to the video conference's path. The path would continue through the network protocol modules (IP, UDP), which would process any header information and possibly defragment IP packets. From here the data would be passed into a video decoder module that would translate the payloads of the network packets into frames of video. These frames would finally be passed to the video display module in order to be viewed by the user.

The path mechanism in Scout allows modules to register interfaces; these interfaces may be used to connect modules together to form paths; this is similar to (but more extensible than) the connectable modules used by Click [Morris99]. A *router graph* is generated to identify valid configurations of connected modules. By associating a particular level of resources with a path – rather than with a network subsystem or the video display subsystem – Scout attempts to offer guarantees to particular activities. A further advantage is that the path abstraction permits further optimisations such as partially evaluating the functions along a path to provide versions specialised for a particular data stream [Mosberger96a]. The performance improvements achieved by these optimisations must be offset by the loss of spatial locality in the processing code caused by multiple specialised versions of common functions.

Nemesis [Roscoe95, Leslie96] is an operating system designed to provide resource guarantees and QoS to applications. It is *vertically-structured* – i.e. wherever possible, applications perform work themselves (typically using shared libraries) rather than calling a shared server to perform the work for them. By making applications less reliant on shared servers, *crosstalk* (interference in the QoS received by one application due to activity caused by another application) is reduced. Where resources need to be multiplexed (e.g. for CPU scheduling and network output), such multiplexing is performed at the lowest possible level, in accordance with the principles put forward in [Tennenhouse89]. Tasks such as protocol processing [Black97], thread scheduling and virtual memory paging [Hand99] are safely delegated to the applications themselves. Nemesis is discussed further in Chapter 4.

2.3.2 Resource Control in Mobile Code and Safe Languages

Existing approaches to resource control in mobile code and safe languages have focused on two main areas: the negotiation for and allocation of resources, and the scheduling and accounting of such resources.

D'Agents [Bredin98] uses a market-based approach, in which mobile agents bid for access to resources. "Sealed" bids allow the resource providers to effectively determine the current demand – and hence market rate – for a resource. [Tschudin97b] describes a similar environment that allows *messenger agents* [Tschudin97a] to bid for resources; actual allocation is performed through *lottery scheduling* [Waldspurger94]. [Lal99] presents a framework to allow mobile programs and hosts to specify constraints on the allocation of CPU. The framework permits the specification of real-time and non-real-time constraints; however, the current implementations do not support real-time scheduling.

Most existing active and programmable network research has tended to ignore the issue of resource control, focusing instead on the development of programming paradigms and environments. Since such work typically occurs in a closed environment, with only a relatively small set of researchers accessing the systems concerned, this has been a reasonable choice to make.

Where thought has been given to resource control, it has typically been coarse grained. For example, PLAN provides three facilities for controlling the execution of a packet's code:

- Associated with each packet is a *Resource Bound* (RB). The RB is an analogue of the IP *hop-count*, and indeed is used in a similar way – each network hop taken by a packet decreases its RB by one, and a packet whose RB is exhausted is discarded. However, use of the RB is more flexible than in IP. If the code for a packet sends out multiple different packets, it may choose how to split its RB amongst the new packets. The RB is also used for evaluating recursive functions – by bounding the number of recursive calls that may be made, a rough bound on the CPU time consumed may be made.
- The total amount of memory allocated by a packet's code at a single node may be bounded, and the packet aborted if it exceeds this bound.
- The total amount of CPU consumed by a packet's code at a single node may be bounded, and the packet aborted if it exceeds this bound.

Safetynet [Wakeman98] uses type information to calculate a bound on the resources consumed by an active packet. A further feature of Safetynet is to draw a distinction between packet-forwarding code and ordinary code. Forwarding code is restricted in what operations it may perform; it may not allocate heap memory, and it must provably terminate. ANTS applies similar restrictions to capsule forwarding code – the `evaluate()` method of a capsule may not call into other Java classes.

The concept of Proof-Carrying Code has been extended [Necula98] to permit formal reasoning about the resource usage bounds for untrusted agents. Whilst these techniques do go some way to prevent serious DoS attacks, they are insufficient for providing effective fine-grained resource guarantees.

More recent work has begun to focus on issues of proper resource control. The specification of the NodeOS [Peterson00a] provides an interface for allocating and scheduling computing and network resources on a programmable node. Early implementations of this interface, such as Bowman [Merugu00], have shown effective isolation between multiple flows of packets passing through a node.

Java has proved a popular medium for resource allocation and control architectures. The original runtime definition contained support for priorities, which by themselves are insufficient to provide effective resource guarantees. J-Res [Czajkowski98] and Conversant [Bernadat98] extend Java's resource control, providing a measure of control over the consumption of resources by Java threads. Both account memory allocations and CPU time to threads, and attempt to provide CPU guarantees by adjusting the priority of threads based on scheduler feedback. A philosophical difference between the two is that J-Res attempts to avoid changing the Java runtime, thus improving portability, at the expense of efficacy, whereas Conversant makes substantial modifications to the runtime in order to provide facilities such as multiple semi-independent heaps.

The J-Kernel [Hawblitzel98] provides support for multiple untrusted threads in a Java system. Different threads are isolated from one another in multiple protection domains within a single heap, but communication is permitted through *capabilities*, which marshal parameters from one domain to another. KaffeOS [Back00] applies abstractions from OS design to a Java virtual machine; threads are grouped into processes, and given independently garbage-collected heaps. Processes may share data in specially designated read-only heaps. The system is divided into "user" and "kernel" sections; user code is untrusted and may be terminated at any time, whereas kernel code is trusted and may not be terminated. Resource consumption by the different processes in a KaffeOS system is accounted, to help prevent denial of service attacks.

2.4 Summary

In this chapter, background information relating to the evolution of programmable networks, resource control, and the execution of untrusted code has been introduced. This provides the necessary basis for the remainder of this dissertation, which presents the argument that the combination of these three areas is vital for the widespread deployment of open mobile code systems. In support of this argument, the RCANE architecture, its prototype implementation, and its evaluation are also presented.

Chapter 3

Resource Control Requirements

This chapter discusses why, and to what extent, resource control of untrusted code is necessary. It considers how the resource requirements of user-supplied code are considerably more complicated to predict than those of packets in a passive network. The types of resources which require control, and the way in which those resources should be accounted, are discussed.

3.1 The Case for Resource Reservations

The question of whether resource reservations or overprovisioning of best-effort services is a more cost-effective way providing services has long been a bone of contention within the networking community. Advocates of reservations claim that modern multi-media applications require the higher predictability given by reservation-capable networks to provide a suitable level of service. Opponents of reservations claim that a reservation-capable network will only provide satisfactory service when its blocking rate[1] is low, and hence the provisioning levels required would provide near-satisfactory service in a best-effort network – the difference being met by a modest amount of over-provisioning, with much lower complexity than the reservation-capable network.

3.1.1 Resource Reservations in Passive Networks

The Internet was conceived as a network giving only best-effort delivery. Protocols such as TCP are designed so that in the presence of contention they throttle back their output, so as not to overload network switches. Token support for QoS was provided in the *Type of Service* (ToS) bit, which allowed packets to be marked according to their traffic type and precedence. Such information is typically ignored by network routers, leading to a best-effort service for all network users. Such behaviour is not always desirable, particularly when attempting to use interactive applcations across a network, or view streamed multimedia

[1]The rate at which it refuses reservation requests

data. Integrated Services (RSVP) [Zhang93] and Differentiated Services (DiffServ) [Blake98] have been two approaches to providing QoS in the Internet (in the case of DiffServ, re-using the IP ToS bit to actually support different service types). More recent technologies such as ATM have provided support for QoS from their inception.

[Breslau98] addresses the question of whether resource reservations are required in passive networks. Different classes of applications and load distributions are considered, and expressions are derived for the additional bandwidth required for a best-effort network to provide equivalent service to a reservation-capable network. No definitive conclusions are presented about whether future networks should be reservation-capable. However, the authors note that the greater the unpredictability of the offered load, the greater the performance advantage of a reservation-capable network over a best-effort network; in particular, with an exponential or algebraic[2] load distribution, the additional factor of bandwidth required by a best-effort network can increase without bound as the base bandwidth increases.

In [Paxson94] traffic traces were studied, with the conclusion that much of the WAN traffic in the Internet could not be modelled with a Poisson inter-arrivals process, but instead exhibited distributions with much larger variances and self-similarity [Leland93, Crovella95]. Although it is difficult to predict the load distributions faced by future networks, such results suggest that reservations will be necessary, at least for certain classes of traffic.

It has been claimed [Deering95] that applications can adapt to whatever service the network offers, rendering reservations unnecessary. Whilst this may be the case, such adaptation can only occur by degrading the QoS delivered to the end user. For some applications that do not require interactive response or low jitter, such as file transfer or electronic mail, this may be a practical solution. However, for multi-media applications adaptation can result in a substantial reduction in the utility received by the end user.

A particular drawback to the argument that adaptation renders reservations unnecessary is that it assumes all communications are of equal importance. This tends to reduce the incentive for users to put off unimportant communication to times when the network is less busy. By allowing (and charging for) reservations, important communications automatically receive improved QoS over less important ones, since their users are likely to be willing to pay the additional cost to ensure high utility.

3.1.2 Resource Reservations in a Programmable Network

For the remainder of this dissertation, the term *open programmable network* is used to refer to a network in which *some* of the nodes provide programmable extensions to end users. To obtain benefits from programmable networks it is not necessary that all nodes within the network are programmable – indeed, since the latency and processing overheads experienced by flows through a programmable platform are likely to be substantially greater than those experienced through a hardware router or switch, providing programmability for that proportion of the traffic that has no special requirements may be considered a waste. The term

[2]An algebraic distribution has a high variance in offered load; the probability that k flows are requesting service is $P(k) = \frac{\nu}{\lambda + k^z}$.

```
let hostileForwardPacket pkt =
    while (true) do
        allocateSomeMemory ();
        sendPacketToNeighbours (pkt)
    done
```

Figure 3.1: A hostile forwarding routine

An example of a forwarding routine potentially capable of consuming all available resources at a node.

open programmable network is used to imply that these programmable facilities are available to all users of the network (possibly in return for a payment), not just those users employed by or trusted by the network providers.

Since providing programmable platforms within the network greatly increases the ways in which end-users may consume resources, it seems reasonable to assume that the variance in load experienced by such platforms will be greater than that experienced by “passive” networks. In a passive network the potential resource load at a node caused by the activities of a particular customer is likely to be roughly proportional to the bandwidth offered to that customer. There is a direct correlation in the case of buffer storage and link utilisation; the CPU cycles required for forwarding a packet are likely to involve a constant per-packet cost for the routing, and a copying cost proportional to the bandwidth[3]. Thus, by limiting the bandwidth that a customer receives, a network provider may limit the amount of resources consumed by that customer on a network node. Effectively, the direct charge that the provider makes for the bandwidth includes an indirect charge for the use of other resources on the network node.

However, in a programmable network there are many more variables to take into account when considering the system load – the loads generated on resources such as CPU cycles, memory and outgoing link bandwidth may be totally unrelated to the incoming link bandwidth. At the extreme, hostile, greedy, or careless forwarding code could potentially consume all available resources at a node. For example, within an active network node, allowing arbitrary customisation of the forwarding code used to process packets would permit (if specific controls were not put in place) a single packet to potentially consume all available CPU, memory and link bandwidth at a node (see Fig. 3.1).

It has been argued [Wetherall99b] that since part of the rationale behind programmable networks is to trade off bandwidth for other computing resources (such as memory and CPU cycles), bandwidth is presumably the scarce resource at the node. Therefore simply scheduling bandwidth still suffices to schedule other resources at the node. However, apart from the obvious problem of gross DoS attacks, there are two flaws in this argument:

- This effectively provides an incentive for end-users to reduce bandwidth requirements at all costs, even if this results in inefficient use of other computing resources. For example,

[3]This is clearly not the case if transport protocols and some of the higher level functions of IP – such as ICMP and subnet broadcast – are taken into account [CERT96, CERT98]. However, such abuses of IP are generally fairly straightforward to detect as DoS attacks, and as such can be filtered out by routers.

a program that is transcoding a multicast video stream before transmitting it over a low bandwidth link would be charged according to the size of its output stream – thus its optimal course of action would be to expend large amounts of CPU making a given quality of video stream fit into the smallest possible bandwidth, even if the majority of that work only produces very small incremental benefits.

- Programmable networks do not just reduce bandwidth requirements at the expense of increased use of other computing resources; they can also reduce latency using the same trade-offs. In this case the bandwidth consumed is potentially a small part of the total resource consumption.

Thus, we conclude that simply limiting the incoming bandwidth is insufficient to allow the provider to control overall resource usage at a node.

Since the load variance in a programmable network may be expected to be greater than that in a passive network, the conclusions of the analysis presented in [Breslau98] may be applied to suggest that some form of resource reservation and control will be required for a programmable network.

3.2 Approaches to Resource Control

[Wetherall99a] recognises that resource control is one of the major unsolved problems in current active network research. The approach taken in [Wetherall99a] is to only accept code certified by a trusted authority – it is acknowledged that this is a serious drawback to the provision of an open programmable network.

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, and/or the (possibly privileged) services which it can invoke. The second requirement is to limit the execution quantitatively – limit *how much* effect its activities may have on the resource usage in the system. Such resource control may be performed in various ways, discussed in the following sections.

3.2.1 Control through proof

Proof-carrying code [Necula97] can be generated; such code can be formally proved to respect the typing disciplines that an execution server wishes to enforce. A verifiable type-safe language represents a subset of proof-carrying code – in this case the server has chosen to enforce the generalised type system used by the language. This, then, provides the *qualitative* control mentioned above. Furthermore some previous work has used such techniques to provide limited *quantitative* control – e.g. using proofs to reason about the number of instructions executed within a particular block of code, in order to show that a routine yields the processor sufficiently frequently to satisfy the execution server [Necula98].

However, such proofs can be expensive to generate and are not particularly general. For example, a block of code that could prove it would yield the processor every 15ms would be unlikely to get a chance to run on a node that had other clients who needed to run every 10ms. Since such proofs are likely to be based on worst-case costs, this could lead to clients being unable to run their code even if it would not interfere with other clients' guarantees. Conversely, if the same code were supplied to a server whose existing client had been guaranteed to be scheduled every 29ms, the 15ms client would get to run only once every 29ms – since it would be impossible to run the 15ms client twice in a row without missing the 29ms deadline. A generalised routine could possibly be developed that took its allowable running time as a parameter, and could be proved to return within a time bounded by that parameter; however, such a routine would suffer both in code complexity and in the overheads required to ensure it did not exceed its limits. Furthermore, when multiple resources need to be controlled simultaneously, such proofs become substantially more complicated.

3.2.2 Control through scheduling/accounting

The alternative to requiring proof is to actively schedule the resouces on the node, and measure the resources consumed to allow accounting to take place afterwards.

Scheduling may be either co-operative or pre-emptive. Co-operative scheduling relies on the code being run to relinquish access to the resource at regular intervals; scheduling decisions may only be made at these intervals. Typically only CPU cycles are scheduled co-operatively, although it could be argued that gaining access to certain system resources that require mutual exclusion (e.g. binding to a specific TCP port, or grabbing the display server focus) involves co-operative scheduling.

Whilst co-operative CPU scheduling may be suitable for multiple threads within certain applications, or – when used with care – within OS kernels, it has proved less suitable for system-wide scheduling, especially in the presence of careless or badly-behaved applications that fail to relinquish access to the CPU; this can result in a system user-interface hanging whilst an application carries out a large calculation, or a real-time application missing its deadlines. A system which permits pre-emption between different clients is essential in order to allow the various guarantees to the clients on the system to be effectively honoured.

It should be noted that for some resources, fully pre-emptive scheduling is not possible. For example, an Ethernet card cannot reschedule more frequently than at packet boundaries. When large (1500 byte) packets are being sent on a 100Mb/s network, this means that scheduling decisions may only be taken at 120μs intervals. (These effects may be lessened by through use of a link layer such as ATM, which by virtue of its small cell size allows scheduling to occur every 53 bytes, equivalent to 4μs on a 100Mb/s link.) In this case, it is necessary to account for any overrun, so that over two or more scheduling periods a client does not receive excessive guaranteed access to the resource due to such quantisation.

3.3 Resources requiring control

Once the decision has been taken to limit the resources available to untrusted code, the resources that need to be controlled must be selected. This section examines the different resources available on a node in a programmable network, and discusses approaches to controlling each.

3.3.1 CPU cycles

One of the most obvious resources to be controlled is that of access to a CPU. This includes time spent executing the untrusted code and time spent executing in (trusted) system libraries. In particular, time required for tasks such as packet processing and garbage collection should be accounted to the principal on whose behalf the tasks are being performed.

Many different algorithms have been developed for scheduling CPU resources. Traditional general-purpose operating systems such as Unix [Ritchie83] use time-sharing schemes that attempt to allocate the available CPU cycles fairly among the processes (or threads) in the system. Each process has a *dynamic* priority that decreases when it receives CPU time, and increases when it is waiting on a run queue; at each point, the process with the highest dynamic priority is run. To improve interactive response and the throughput of I/O bound processes, such processes are given temporary boosts in their dynamic priority. Time-sharing schemes make no guarantees that a given process will receive access to the CPU with any degree of timeliness.

In an attempt to provide more reliable access to the CPU for processes that require it, some schedulers introduce "real-time" *static* priority classes – at any point in time, the scheduler runs the process(es) with the highest static priority. If a low-priority process is running and a high-priority process becomes runnable, it typically preempts the low-priority process. If several processes share the highest priority value, they may be scheduled cooperatively or preemptively.

Priority-based schemes have several flaws when seeking to provide resource guarantees [Nieh94]. The first is that a priority is by definition a relative value – the amount of time received by a process of a given priority is very dependent on the number of other processes with equal or higher priority. Thus although a process may experience good access to resources when it has the highest priority in the system, clearly only a few such process can exist before the problems associated with standard time-sharing occur. The second flaw is that priorities are insufficient to deal with the case of a process that requires regular processing time, but which does not require priority over other tasks whenever it is runnable. In a programmable network, an example of such a task would be the processing of routing updates – these should clearly not take priority over jitter-sensitive tasks; however, it is vital that they get to run for a certain amount of time during every update period.

An approach to scheduling that attempts to ensure that all processes make progress, whilst still allowing straightforward relative priorities, is *lottery scheduling* [Waldspurger94]. Each process is allocated a number of "tickets" according to some resource allocation scheme. At each time slice, a process is chosen at random to receive CPU; the probability distribution used is weighted by the number of tickets held by each process, so a process with more tickets receives more CPU time. This provides a very simple scheduling algorithm that is responsive to users needs; however, since no guarantees are made to processes about when they will receive the CPU over short periods of time, it is unsuitable for jitter-sensitive activities.

The concept of *processor bandwidth* allows processes to receive the guarantees that they require to carry out jitter-sensitive tasks. At its most basic form, processor bandwidth may be expressed as a percentage of the CPU. Whilst this is useful in itself, it does not take into account the fact that different tasks will have different requirements for the frequency at which they receive access to the CPU. [Black94] presents a scheduler that divides up the available CPU time into fixed length periods of time called *jubilees*. Within each jubilee, time is allotted to all processes according to their allocated percentage share. This provides a very simple run-time scheduling algorithm, but requires that all processes be scheduled at the same frequency (i.e. the inverse of the jubilee length). Thus the presence of a process that requires frequent scheduling causes all processes to be frequently scheduled, increasing scheduling overheads. In the presence of processes with very different requirements for their scheduling frequencies, a better solution is to guarantee each process a certain fraction of the CPU over a process-defined period of time; thus the guarantee consists of a pair (p, s), to allow a process to receive p seconds of processor time in each consecutive interval of length s seconds.

The concept of periodic *deadlines* may be employed to provide the appropriate regular access to the CPU required by processor bandwidth scheduling schemes. [Liu73] describes the Rate Montonic (RM) and Earliest Deadline First (EDF) algorithms for scheduling hard real-time tasks. RM calculates static priorities for each task, according to the frequency at which they must run; at each scheduling decision, the runnable task with the highest priority is executed. EDF effectively recalculates such priorities dynamically, always running the task with the closest deadline. EDF has a greater runtime complexity than RM; however, [Liu73] shows that EDF can give a feasible schedule for any utilisation up to 100%, whereas RM can only guarantee a feasible schedule up to 69% ($ln2$). EDF and RM are both designed for the scheduling of periodic tasks, rather than general purpose process scheduling. [Roscoe95] shows that by considering the guarantee made to a process as being a periodic task requiring s seconds to run and with a deadline every p seconds, these algorithms are applicable to the problem of processor bandwidth scheduling.

3.3.2 Memory Usage

Memory usage by untrusted code falls into five areas:

3.3.2.1 Code modules

These represent code that has been dynamically loaded, either at system initialisation time or by remote clients. They are likely to be read-only[4], allowing a single module to be shared amongst multiple clients. A code module may have some form of persistent variables.

3.3.2.2 Thread stacks

Thread stacks provide the context associated with a computation occurring on the node. This may be in order to forward or otherwise process a packet, or it may be in response to some other event, such as a timer expiring. Stacks represent a measure of concurrency – if blocking (or long-running) computations are expected to occur, the availability of thread stacks for a particular class of remote client limits the response time and processing rate for that class of clients. Examples of such classes might include a single client with resource reservations, or the set of “all best-effort clients”.

3.3.2.3 Packet buffers

Packet buffers provide a level of queueing for both transmission and reception at network interfaces. If a client had been guaranteed exclusive access to the CPU, and if the latency in the paths between the network device drivers and the client were negligible, such buffering would not be needed. However, neither of these conditions are likely to be met; thus to be able to keep a transmission queue fed and avoid dropping incoming packets in the periods between gaining access to the CPU, network buffering must be provided.

3.3.2.4 Heap memory

For some forms of communication in a programmable network, each packet may be totally independent and not rely on flow-specific information stored at nodes in the network – such flows could include datagram flows that consulted alternative routing tables, or capsules that carried all their code and data with them. However, carrying multiple copies of the same code and data around a network wastes bandwidth and processing time (for marshalling and unmarshalling between wire format and memory structures), and reduces the amount of useful payload in each packet.

Various approaches have been taken to provide persistent state at network nodes. The most common approach has been to provide some form of *soft state* – data that may be flushed from storage by the node if it requires the space for something else. Thus the client cannot rely on this data remaining available between successive packet arrivals. Examples of this form include:

[4]Whilst self-modifying code does have its uses, we regard its deployment in a programmable network as foolish, due to the extreme complexity of debugging such code remotely and the prevention of sharing that such modification entails.

- ANTS' [Wetherall98] code cache, in which the Java classes for a particular packet type are cached along the path taken by those packets.
- PLANet [Hicks99c] and Bowman [Merugu00] provide timed soft-state stores.

ANTS is structured such that in the event of an arriving packet's protocol code having been flushed from the cache, it can be easily retrieved from the upstream node that last processed the packet.

However, other kinds of data (such as flow routing information) may not be so easily regenerated. Furthermore, when all packets are processed by the same set of threads – i.e. without reserving stack resources for particular remote clients – the only way to access per-client information is to store it in some form of lookup table. For example, PLANet's soft-state mechanism uses a table keyed on client identifier (for security) and a string supplied by the client (to permit multiple data items to be stored by the same client). This represents inefficiency, a possible source of crosstalk (in the table locking) and an unnatural style of programming.

For clients with reservations, an alternative is available – the code environment used to process packets and events for a client may be specialised to that client; therefore instead of invoking a general packet processing routine that locates an appropriate function to call for this flow, and provides access to some form of soft-state lookup, instead the function specified by the client can be invoked directly, with language-level access to persistent variables defined by that client's code.

In order to store such data, heap space must be reserved for that client. It should be possible to limit the amount of heap space consumed by a particular client; conversely, it should be possible to guarantee to that client that they will receive the amount of memory that they have reserved.

3.3.2.5 Auxillary data structures

The system will need to maintain data structures concerning clients on the node. These may be for administrative purposes (e.g. billing and accounting information) or for functional purposes (e.g. structures relating to network connections). These structures are unlikely to consume large amounts of memory during normal execution; however, the possibility of a buggy or malicious client means that it is necessary to limit this memory consumption. Such structures are likely to fall into three classes:

- Structures of which a single instance exists per client (e.g. client identity). These do not present a resource control challenge, since there is no way for a malicious client to increase the resource usage associated with them.
- Structures that are associated with controlled resources. For example, a thread control structure is going to be allocated in association with a thread stack; a network control structure is going to be allocated in association with network buffers and a bandwidth

reservation. These structures do not need to be controlled individually – it will suffice to include their effects in the accounting for the controlled resources.

- Structures that may be created through client actions and which are not associated with controlled resources (e.g. timer events). These should be accounted for in some way. Two approaches are to track the allocations of each individual type of objects; or to maintain a (fixed-size) per-client heap from which all such allocations are made.

3.3.3 Network Bandwidth

In a node (switch or router) in a traditional passive network, the bandwidth offered to a network flow is one of the fundamental metrics of quality of service. In a programmable node it is quite likely to be secondary to other resources such as CPU cycles or memory – if the client purely wanted network bandwidth, it is likely that reserving resources on a programmable node would incur greater overheads (and hence be more expensive) than setting up a flow through a standard switch or router. However, the ability to reserve network resources on a programmable node is still vital; consider the situation in which a client has moved its logic for a networked transaction to a node closer to the server, in order to reduce the latency of the transaction. If the latency between the client and the server is 100ms, and the client's code running on the network node only gains transmit access to the network every 200ms, the latency of the transaction will be increased rather than decreased by the migration.

Various transmission scheduling algorithms have been proposed to share out network bandwidth. These include *weighted fair queuing* [Demers89] and *virtual clock* [Zhang90] and a modified form of EDF [Black97].

3.3.4 Summary

This chapter has examined issues relating resource reservation in programmable networks. First the arguments for and against resource reservations in passive networks were considered. By comparing the likely load variance at programmable nodes with that of passive nodes, it was concluded that resource control in programmable networks would be required.

Two approaches to resource control were considered – control through proof and control through scheduling. It was concluded that proof-based controls were useful for enforcing safety properties, but for ensuring guaranteed access to resources they were less practical than scheduling.

The chapter concluded with a discussion of the resources on a programmable node that would require control.

Chapter 4

Nemesis

This chapter presents further detail on the Nemesis Operating System developed at the University of Cambridge. The origins of Nemesis are discussed, along with the properties that make it suitable as a base for the RCANE architecture to be presented in Chapter 5. More information may be found in [Roscoe95, Leslie96, Barham96].

4.1 Overview

In traditional operating systems, tasks such as network protocol processing, paging, and graphics rendering are performed either in a shared server or in the kernel – unless this work is carefully scheduled and accounted for, it can cause Quality of Service interference (crosstalk) between applications.

Nemesis simplifies such accounting by requiring each client to perform much more of its own work than in a traditional system; shared servers (or the kernel) are needed only to arbitrate access to shared resources. Wherever possible, these servers exist purely for setting up and destroying access connections (e.g. providing access to the blocks on a disk consituting a file, or installing a packet filter to demultiplex network packets). Other operations such as network protocol processing or graphical rendering are performed in the application itself (typically using shared libraries), with the servers just providing safe scheduled access to the hardware.

This approach allows Nemesis to provide an effective platform for "multi-service" systems, in which different multi-media applications co-exist without experiencing interference from one another or from other applications with less critical time constraints.

4.2 Operating System Structure

4.2.1 Traditional Systems

Traditional operating system structures have fallen into three main categories:

Monolithic systems have a single operating environment that has full access to the machine's resources. Typically only a single application runs at any one time, or else co-operative task-switching may be employed.

Kernel-based systems separate out applications from system services (scheduling, memory protection, paging, scheduling, etc). Services run in a privileged *kernel*; applications run without system privileges, and isolated from each another.

Such systems are typically not designed with the intention of accurately accounting the consumption of resources to applications. Substantial amounts of work are often performed within interrupt handlers, such as network receive routines; accounting such work to the receiver can be problematic.

A further problem with the kernel-based approach is that the abstraction layer and the protection layer are entwined; whilst a high level of protection is desirable, the high abstraction level thus presented prevents an application from effectively specifying the relative importance of its own activities.

Although some research has addressed these issues [Druschel96, Banga99, Ingram99, Bruno99], effective resource control in traditional kernel-based systems is still an open problem.

Microkernel systems aim to provide modularity (and thus greater maintainability and fault isolation) by moving functionality from the kernel into servers. The kernel itself provides support just for scheduling, memory protection and Inter-Process Communication (IPC); device drivers[1] and services such as networking, paging, filesystems and memory management are provided by (possibly privileged) servers.

Whilst this approach does provide a more flexible method of OS design, the additional modularity and protection can increase the overheads of communication between applications and the various servers. However, previous work [Härtig97] has suggested that with careful optimisations, the cost of such communication can be comparable to a traditional kernel-based system.

A more fundamental problem with the microkernel approach is that it makes accurate resource accounting even more difficult than in the case of a kernel. Let us take the example of a network stack server. If it is to provide QoS, it must schedule the actual transmission of outgoing packets and account for the buffering of incoming packets. However, if it is also performing protocol processing (such as checksumming, IP fragmentation/reassembly, etc) on the data being transmitted and received, it must also have sufficient CPU resources to carry out such work; furthermore the server must be

[1]In some situations, portions of device drivers are placed within the microkernel for performance or other reasons.

able to accurately account such resource consumption to the client applications. Similar problems apply with the rendering time used by a graphics server[2] and the disk bandwidth consumed by a paging server.

4.2.2 Vertical Structure

A new approach to OS structure, taken by Nemesis [Leslie96] and the Exokernel [Engler95], is the *vertically-structured* system. Vertically structured systems arose from the concept that the association between abstraction and protection enforced by kernel- and microkernel-based systems was unnecessary. In fact, while enforcing a high level of protection on applications is desirable, enforcing the high level of *abstraction* is not.

In a vertically structured system, many tasks traditionally performed by the kernel or shared servers are migrated into the application itself. The high level of abstraction provided by traditional systems is instead provided by shared libraries; since the work is performed by the application itself it can be easily accounted to that application.

Nemesis and the Exokernel approach the concept of a vertically structured system from different directions – the Exokernel is essentially a kernel-based system and Nemesis a microkernel. However, in both systems, all but the control path activities (such as setting up IPC connections and specifying network packet filters) and very low-level data path operations (such as transferring individual disk blocks, ethernet frames and pixel blocks) have been moved into the applications themselves; thus the work remaining for the kernel or the shared servers is minimal and does not significantly contribute to QoS crosstalk.

In the following sections, various aspects of Nemesis are described to illustrate the vertically structured nature of the OS, and show why it is suitable as a base for a programmable network node.

4.3 CPU Scheduling

Nemesis allows a domain to reserve a single guaranteed allocation of CPU time, in the form of a slice of CPU received over a period of time. The scheduling period may be very fine-grained, and periods as short as a millisecond are common for processes that require low-latency. The current implementations of Nemesis use the Earliest Deadline First (EDF) [Liu73] scheduling algorithm to allocate CPU time.

When an application becomes eligible to receive CPU time, it is entered at its *activation vector*; this will typically handle any incoming events (see Section 4.4) and then dispatch to a user-level thread scheduler. This gives the application full control over how its allotted CPU time is used.

[2]The problem of accounting for time consumed by a graphics server has also been seen to occur with the *X* server on Unix systems [Barham96].

Because Nemesis requires that applications perform their own work rather than relying on servers, it is possible to significantly regulate the total CPU load put on the system due to an application's activity. Furthermore, since far more of the activities accountable to an application take place within that application's protection domain (rather than in a kernel or server) the application has far more control over the scheduling of its own tasks. For example, an application may choose to expend time on protocol processing for a high importance network stream, and defer any processing for a low-importance stream. Such behaviour cannot be easily achieved in a traditional OS, although projects such as *Signalled Receiver Processing* [Brustoloni00] and *Resource Containers* [Banga99] have addressed this problem for monolithic kernel-based systems.

4.4 Interprocess Communication

The Nemesis kernel provides only the simple *event channel* primitive for communication between *domains* (Nemesis processes). An event channel is a monotonically increasing integer shared between a sender and a receiver; when the sender increases the value of the event, the new value is made available to the receiver, and the receiver is optionally notified of the change and unblocked if necessary.

On platforms in which transitions to kernel mode may be made cheaply, such as the Alpha 21164, such event increments are performed through system calls; on platforms such as Intel x86 and StrongARM, where kernel transitions are more expensive, a per-process FIFO of outgoing event increments is read by the kernel at reschedule time.

Higher levels of IPC may be built on top of these primitive events by applications, with no additional support required from the kernel. A system service called the *Binder* allows domains to establish event channels to other domains.

The standard Nemesis libraries provide synchronous closure-based IPC using shared-memory segments. When an IPC invocation is made, the arguments to the invocation are marshalled by the client into the shared-memory buffer; on the server side the arguments are copied out of the buffer in order to make an invocation on the original closure.

This marshalling is performed by *stubs* that are generated dynamically, at runtime; when the first IPC channel of a particular interface type is created, the methods in that interface are analysed [Menage98a]. For each method, a bytecode description of the *actions required* to marshal its parameters is generated; from this a machine code stub is synthesised. Such stubs are cached, and reused later if a method with an equivalent bytecode sequence is processed. The low-level bytecode used to describe marshalling actions allows significant code sharing to occur between stubs.

A second form of IPC is the *I/O Channel* [Black94, Barham96]; this supports buffered asynchronous communication between domains using three areas of shared memory. The first area of memory contains the data buffers being transferred; the second is used as a FIFO from the consumer to the producer, for passing descriptors indicating empty buffers (within the data area); and the third is used as a FIFO from the producer to the consumer for passing descriptors indicating full buffers. Event channels are used to synchronise access to the FIFO areas.

4.5 Device Architecture

Nemesis uses the DMM/DAM device architecture proposed in [Barham96]. Control over each device is split into two independent entities[3]:

- The *Device Abstraction Module* provides data-path access to the device. Connections (typically abstracted behind an I/O channel) are established by the DMM, over which clients may send and receive data. These connections may be implemented in several ways:
  - If the device is a *user-safe* device [Pratt97], the client may be given protected direct access to a set of registers on the card; thus the I/O channel encapsulates the logic for communicating buffer areas to and from the card.
  - If the dominant resource consumed when accessing the device is CPU time, and requests to the device may be broken up into small units, a trap into kernel mode (known as a device privileged section or *callpriv* [Barham96]) may be provided by the device driver to access the device. For example, Nemesis frame buffer drivers use a callpriv to allow clients to transfer small blocks of pixels into windows that they own. Since the dominant resource consumed is CPU cycles (due to the relatively low speed of the PCI bus), the CPU scheduler effectively polices access to the device with no additional scheduler required in the framebuffer device driver.
  - If the device has the ability to schedule between multiple clients, but cannot provide protection between them, a callpriv may be used to provide the necessary protection to make the device appear to be a user-safe device. For example, the DEC OPPO ATM adaptor has the ability to create multiple outgoing queues of packets, each separately scheduled. For this device, a callpriv is used to mark an outgoing PDU for DMA and transmission on the client's queue.
  - For traditional devices that support neither scheduling nor protection, a thin device driver layer is required to mediate access to the device. This layer should provide access in terms of small requests (such as disk blocks, network frames or tiles of pixels) and may be required to perform scheduling for clients.
- The *Device Management Module* (DMM) provides management and control for the device; it is responsible for implementing access control and resource allocation policies, and for creating data connections between clients of the device and the DAM. It takes no part in data-path activity.

[3]Note that in some cases it proves convenient to implement these two entities as two interfaces on the same object; however, any synchronisation between the two is structured so as to avoid interfering with data-path guarantees.

4.6 Network I/O

Nemesis provides network access in the form of I/O channels between applications and the network device or device driver, over which link-level frames (such as Ethernet frames or AAL5 PDUs) are exchanged. The application is responsible for performing any necessary protocol processing (such as flow control, reliable delivery, IP fragmentation and reassembly, or TCP checksumming and segmentation); the device driver needs only to schedule the transmission and reception of raw frames. Packet filters are used both on transmission (to ensure that the packet headers are valid) and reception (to locate the I/O channel to which the packet should be passed).

The Nemesis *Flow Manager* is the DMM for the network – it is in charge of setting up the I/O channels between clients and network devices and configuring packet filters; any required network access policy for the system is implemented by the Flow Manager. However, once such I/O channels have been set up, the Flow Manager plays no part in data-path activity.

In the case where the network interface card is a user-safe device, the client application may communicate directly with the card. For traditional devices that cannot themselves provide protection between multiple users accessing the device, a thin device driver layer is needed. This driver is responsible for scheduling and transmitting Ethernet frames from a set of client I/O channels, and for receiving and demultiplexing incoming frames, and sending them down the appropriate I/O channel if that channel has empty buffers available. In the event that an incoming I/O channel is full, the packet is dropped, either in the card or in the driver before protocol processing takes place.

Since each I/O channel acts as an independent queue, and processing time is only expended on processing packets in that channel when the application so chooses, accurately scheduling and accounting multiple incoming or outgoing network streams in a single application is more straightforward under Nemesis than under traditional systems. For example, a web server could give priority to channels that corresponded to sites hosted for paying clients, and only expend resources on non-paying clients after all work for paying clients had been performed.

4.7 Summary

This chapter has provided detail on the Nemesis Operating System. An overview of the structure of the OS was presented, with a comparison to the structures of traditional operating systems.

Aspects of the system, including its device architecture, inter-process communication models and resource scheduling models were presented; it was shown that Nemesis possesses features that support effective resource isolation between multiple clients, and provides the ability for applications to schedule the resource consumption of their own activities. These properties suggest Nemesis as a suitable base for the development of a platform for the execution of untrusted code in a programmable network.

Chapter 5

The RCANE Architecture

This chapter describes the Resource Controlled Active Node Environment (RCANE) architecture and the principles guiding its design. RCANE is designed to allow providers of nodes in a programmable network to permit untrusted clients to run code on their nodes, without the risk of Denial of Service attacks or excessive consumption of resources.

RCANE provides abstractions to control access to CPU cycles, network bandwidth and memory, and allows lightweight and flexible communication between clients. The structure and mechanisms used by RCANE to accurately account for these resources are described, and their applicability to preventing various types of DoS attack is discussed.

Many of the ideas presented in this chapter were previously described in [Menage99].

5.1 Introduction

In the previous chapter the Nemesis Operating System was discussed. It was shown that Nemesis provides basic resource guarantees to applications, allowing multi-media applications to co-exist with one another and with batch processes without the interference associated with traditional operating systems.

Given the resource requirements discussed in Chapter 3, this would seem to make Nemesis a suitable platform for a programmable network node. However, in its basic form Nemesis is unsuitable for such a rôle:

- Providing each remote client with their own (hardware protected) Nemesis domain would be expensive in terms of both memory consumption and setup time. The increased memory costs would reduce the scalability of the system. Many clients' applications (such as mobile agents) in a programmable network will be substantially more ephemeral than multimedia applications on a workstation, and so the increased setup costs can turn into a substantial overhead.

- Nemesis is designed to allow essentially benign applications on a workstation to communicate their resource needs (possibly overriden by the user) to the OS and for such needs to be effectively met by the OS. In an uncooperative environment such as a programmable network, applications may be greedy or even malicious, and hence more control is needed.
- Nemesis is designed to run native code. In a heterogeneous programmable network, unless we wish to burden clients with the task of producing and shipping multiple binary formats – possibly for architectures for which they have no suitable compiler – a common virtual machine layer is required over the OS.

The Rcane architecture, presented in this chapter, meets these requirements.

5.2 Architectural Principles

Rcane is designed to provide resource isolation between multiple independent applications on a node in a programmable network, with the resources consumed by each application being paid for by a remote principal. The following aims underlie the design of the architecture:

- To provide guarantees to applications that they will receive the Quality of Service that they require in order to complete their tasks in a timely manner.
- To accurately account resource consumption to the client that causes such consumption to occur, in order that the client may later be be billed.

This section describes how these aims have shaped the overall architecture of Rcane.

5.2.1 System Structure

Rcane employs both horizontal layering (between different layers of trust) and vertical isolation (between different clients).

5.2.1.1 Layering

Rcane follows the principles (proposed in [Alexander98a]) to partition the system into multiple layers:

- The *Runtime* is written in unsafe native code and provides access to – and scheduling for – the resources on the node. Services such as garbage collection (GC) and thread synchronisation primitives are also provided by the Runtime.

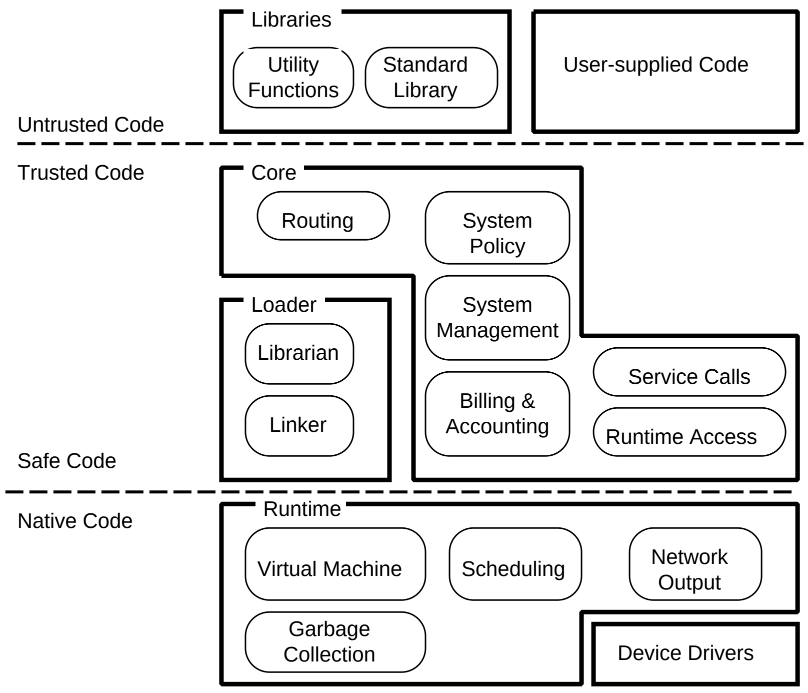


Figure 5.1: Rcane Architecture Overview

- The *Loader* is written in a safe language (as are all higher levels). The Loader is entered early in system initialisation. It is responsible for:
  - completing the initialisation of the Runtime,
  - loading the higher levels of the system, and
  - linking user-supplied code into the system.
- 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. The interface to the Core represents the "Red Line" identified in [Back99] as a requirement for security in a system using software protection.
- *Modules* are units of untrusted code. The include standard *libraries*, supplied by the system and loaded at system initialisation time, and code supplied by remote users. They have access to the interfaces exported by the Core, but no direct access to the Runtime or the Loader except where permitted by the Core.

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 modules can link directly. The Runtime performs a policing function on the use of the node's resources. An overview of the Rcane architecture is shown in Figure 5.1. The Safe/Native code boundary indicates the division between unsafe native code (written in a language such as C) and code written in a safe language supported by the Runtime's virtual machine. The Trusted/Untrusted code boundary indicates the division between code that is known to respect the security properties of the node and other code; such code may be regarded as untrusted if it is supplied by an untrusted source, or if it is from a trusted source but has not been sufficiently checked to ensure that it would not compromise the system. Within the Core and the Loader, although all code is written in a safe language, the interfaces exported by the Runtime permit complete control over the node[1]. Thus it is important that malicious code not be permitted to execute within the Core.

5.2.1.2 Sessions

RCANE uses the abstraction of a *Session* to represent a principal with resources reserved on the node. A session is the analogue of a *process* in a conventional OS that uses hardware protection, and is similar to the concept of a *flow* in the Active Networks NodeOS [Peterson00a]. Sessions are isolated, so that activity occurring in one session cannot prevent other sessions from receiving their guaranteed QoS, except in situations where explicit interaction is requested (e.g. due to one session using services provided by another session).

To provide guaranteed levels of QoS to remote principals, RCANE allows sessions to reserve resources in advance. Requests for resource reservations are processed by the System session (see below) 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.

In other active network systems, the main resource principal is the execution environment (EE). This can lead to QoS crosstalk between the different clients of an EE. The use of sessions in RCANE makes the end-user the resource principal, allowing guarantees to be made more easily to individual end-users. An EE then becomes a library that a session may use to provide a convenient programming abstraction, and a client may make use of more than one EE in a single session if desired. (See Section 5.6 for a discussion of mechanisms allowing shared state between multiple instances of the same EE.)

Figure 5.2 shows part of the `Session` interface provided by the Core to permit control over a session and its resources. (Certain functions concerning modification of resource requirements are not shown.)

- `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 module to be linked may be

[1]In particular, some of the low-level features of the Runtime may permit language safety to be compromised.

```
bool createSession (c : Credentials, r : ResourceSpec, code : CodeSpec);

void destroySession (void);

bool loadModule (l : LoadRequest);

bool linkModule (l : LinkRequest);

vp_id createVP (spec : CpuSpec);

bool bindDevice (d : Device, bu : BufferSpec, bw : BandwidthSpec);
```

Figure 5.2: The `Session` interface

specified simply by the interface that it exports, or by a digest of the code implementing the module to ensure that a particular implementation is used.

- `createVP()` creates a new scheduled allocation of CPU time (see Section 5.3).
- `bindDevice()` reserves bandwidth and buffers on the specified network device (see Section 5.4.

At system initialisation time two distinguished 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. Such system services include:
  - The code librarian and linker.
  - The session manager.
  - The default network routing tables.
- 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 very restricted language and are given minimal access to system resources. Access to the `createSession()` call in the `Session` interface is permitted, so that best-effort code can initiate a new session; further work may then be performed by the newly created session.

Figure 5.3 shows how RCANE sessions are orthogonal to the layering described in Section 5.2.1.1. The horizontal dashed lines indicate boundaries of *trust*; the vertical dashed lines indicate boundaries of *resource isolation*. It can be seen that some portions of the Core – such as those

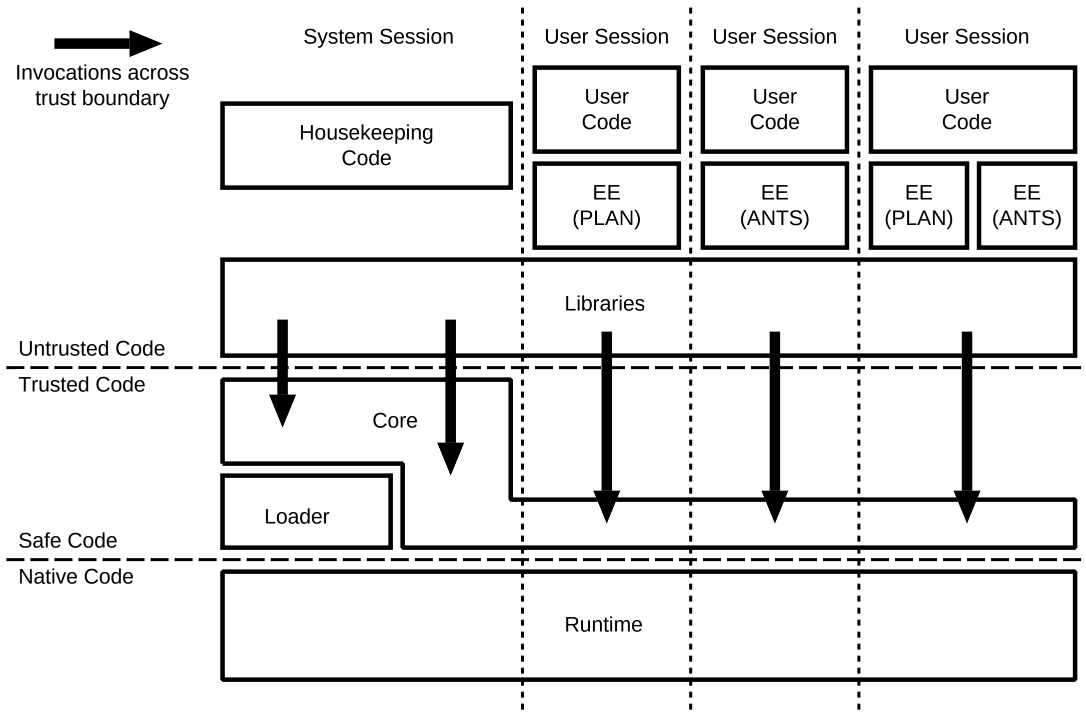


Figure 5.3: Orthogonality of sessions and layering in RCANE.

dealing with data-path activity – are directly accessible to user sessions; untrusted code may call them directly, possibly resulting in a direct call to the Runtime. The majority of the Core code executes in the System Session and thus is not directly accessible to the user sessions; such separation may be achieved using a heap isolation technique such as that described in Section 5.5. Hence this portion of the Core must be accessed through the inter-session services described in Section 5.6.

It can be seen that each user session has created an instance of either the PLAN [Hicks99c] or ANTS [Wetherall98] execution environments, or both. Note that although multiple sessions are using each EE, these instantiations are operating independently and without QoS crosstalk, since resource scheduling occurs in the Runtime, below the level of the EEs.

5.2.2 Security

When opening up a node to permit the execution of untrusted code, security is clearly an important concern, with two important aspects that need to be considered.

The first aspect is that the server may not trust the code supplied by remote users. Therefore, the server must restrict the ability of the user-supplied code to interfere with the code or data of the server or of other users. Such restriction takes two forms: restricting the basic operations – at the language/instruction level – that the user's code may perform (discussed in Section 5.2.3) and restricting the higher-level services that may be accessed.

To identify which services a client is authorised to access, some form of security and trust management framework is required. Rcane does not mandate any one particular security framework, and detailed discussion of such is beyond the scope of this dissertation. Possible frameworks suitable for specifying and implementing security policies in a programmable network environment are presented in [Blaze99, Hayton96, Hicks99b].

The second aspect of security that needs to be considered is that the "untrusted" mobile code may itself not trust the server. The server has total control over the environment in which the mobile code executes; the server is at liberty to examine or corrupt any of the client's code or data, and can adversely affect the execution of the client's code. Once the code is executing on the server the results of an attempt to verify the credentials of the server cannot be trusted.

A possible way to get around this problem is for the algorithms and data used by the client's code to be encrypted in some way; thus the server executes the algorithms, but has no way of understanding what the client is actually doing. This is similar in principle to the *Chinese Room Argument* presented in [Searle80], in which a person carries out a set of instructions to manipulate Chinese symbols, in order to take part in a (written) dialogue in Chinese without understanding either the questions or the answers. [Sander98] discusses mechanisms through which a client's code may provide itself with limited protection against a malicious server, and [Karjoth98] presents a protocol by which a mobile agent's owner may detect alterations made by a malicious host to information gathered from other hosts; however, it is not clear whether these approaches can be extended to provide a general protection mechanism for a client's code.

Therefore, a tenet of the Rcane architecture is that the client supplying the code trusts the server executing the code – this is similar to the requirement in the current Internet that hosts trust the routers in the intervening network path, and does not preclude the use of end-to-end encryption on sensitive portions of the payload that are not needed within the network. Mechanisms for establishing such trust are beyond the scope of this dissertation. An example of an active network architecture in which establishment of mutual trust is a fundamental principle is given in [Alexander98c]. [Bos99] presents an interface through which roaming agents may establish such trust.

Security of a client's data whilst the data is travelling over a network to/from an Rcane node is to some extent the responsibility of the client. Rcane does not specify any secrecy, integrity, or authentication mechanisms for network flows. Rather, it provides low-level access to the network over which the client may layer any desired security mechanisms before ultimately processing the data received in a network flow.

5.2.3 Code Safety

Since an active node is expected to execute untrusted code, there needs to exist a layer of protection between each principal and the node, and between multiple independent principals.

One approach to such protection is to utilise the memory protection capabilities of the node's hardware. Each principal's code will execute within its own address space. This allows the execution of programs written in arbitrary languages and compiled into native code. The virtual machine and interfaces provided to permit a principal's code to interact with the node and with other principals on the node may take several forms.

The server may present a virtual machine that emulates the real machine [Goldberg74, Bugnion97, Creasy81] or provides a trap-based interface to a well-defined kernel interface. Advantages of these approaches include the fact that arbitrary optimisation techniques may be employed by the prinicipal supplying the code in order to improve efficiency of execution. Disadvantages include the cost of context switching between multiple protection domains, and the storage overheads associated with maintaining each domain.

An alternative is to require principals' code to be written in a typesafe and verifiable language. Such languages prevent the programmer from circumventing the type system of the language by treating an object/value of one type as though it had a completely different type. Moreover, programs are either supplied in source form or (more usually) compiled to some intermediate form[2] that the server can be confident respects the type system of the language. This allows much of the protection checking to be done statically at compile or load time, and allows much more lightweight barriers between principals. In particular, it means that untrusted code may be safely executed in the same address space as the RCANE runtime system, and interactions between principals can be made almost as efficient as a direct procedure call.

The constraint of verifiability requires principals to supply code in one of three forms:

- Verifiable bytecode. This may be executed by an interpreter without further checks, or converted into native code by a just-in-time compiler.
- Unverifiable bytecode or scripts that must be run by an interpreter with dynamic checking. This is likely to run substantially more slowly than verifiable bytecode; however, the startup costs of such code will be lower than that of code that must be verified.
- Proof-Carrying Code: native code, with a proof that the code respects the typing and the interfaces provided by RCANE [Necula98]. Proof-carrying code requires the client to generate and prove extensive verification conditions, to demonstrate that the code respects the interfaces provided by the host. These proofs can grow exponentially in the size of the code being proved safe.

Hybrid forms are possible; for example in Java bytecode, some instructions (such as certain dynamic casts) have to be checked at runtime, while other parts of a Java program can be compiled to native code and run without checking.

A major advantage of using bytecode is that in a heterogeneous network it removes the need for principals to supply native code specific to the architecture over which the node is running.

Executing bytecode is somewhat slower than executing native code; however, the performance improvements available when clients supply native code are limited for two reasons:

[2]It is quite possible for multiple source languages to compile to the same intermediate form (as has occured with Java bytecode [Gosling95a]), and thus such an approach need not overly restrict the freedom of the programmer.

1. Just-in-Time compilation [Cramer97] (JIT) techniques allow a bytecode sequence to be converted into equivalent machine code at runtime, potentially providing performance at the same level as that obtained from unsafe code. (For code that is migrating frequently, the overhead of JIT-compilation may be wasted, since the code will not be heavily executed at any one node – the techniques proposed in [Harris98] may be used to permit user-control of such compilation).

2. Code provided as part of the system is trusted, and hence may be fully compiled to efficient native code when the system is created/installed. If the set of services and libraries provided by the node is sufficiently encompassing, most execution time will be spent in these system libraries, and the efficiency of the user-supplied code will be less relevant.

Three other possibilities for supplying safe code were considered but rejected:

1. Software Fault Isolation (SFI) [Wahbe93] would provide a guarantee that the application was not able to access objects to which it did not have any access; however, it would be unable to guarantee that the application respected the type safety of objects within its heap. This would result in Rcane being unable to store any data within the application's heap, nor trust the validity of abstract objects passed to and from Rcane interfaces. Furthermore, [Seltzer96] relates that the runtime checking overhead associated with SFI can be very expensive for data-intensive code.

2. Code could be certified by the compiler as being a true and valid compilation of a program written in the safe source language. Such an approach has two drawbacks:

    - It requires all programmers to use the same trusted compiler, or else requires a trusted party to check the validity of third-party compilers and produce certifying versions of such. The use of verifiable code has no such requirement, since anyone may produce a compiler that generates verifiable code.
    - Unless a small set of trusted authorities provide a "signed compilation" service through which all programs must be sent before being deployed in a programmable network, it would be necessary for the trusted compilers to be distributed to end-users. In this situation it would be difficult to prevent reverse-engineering of the compilers to retrieve the cryptographic keys used for code certification. This is shown by the case of the DVD Content Scrambling System (CSS). CSS is an encryption system intended to prevent the playback of DVD films by unlicenced programs; the keys required to decrypt the per-disk key (which in turn could decrypt the media content on the disk) were embedded in licenced viewer programs. Since the viewer programs were run in a hostile environment (i.e. a system controlled by an end-user) the keys were relatively easily retrieved through reverse-engineering.

      Any malicious user who gains access to the keys may sign arbitrary (possibly invalid) programs. The use of verifiable code does not suffer from this drawback, since the code can be verified on its own merits, without requiring any special key embedded within the compiler.

3. Code could be certified as being provided by a trusted source (rather than, as in the previous item, verified code from an inherently untrusted source). This removes the need for protection, but instead relies on the owner of the node trusting every user who may wish to execute code on the node[3]. In an open network this requirement is likely to be unsatisfiable.

For the reasons given above, RCANE is oriented towards a node supporting the execution of verifiable bytecode or proof-carrying code; however, many of the principles guiding the design of RCANE could be applied to a system that permitted the execution of arbitrary native code.

5.2.4 Modules

The unit of code linkage in RCANE is the *module*. Each module exports a (possibly empty) *interface*, and imports a set of interfaces from other modules. An interface is a strongly typed set of functions and values. Multiple modules may export different implementations of the same interface. Each implementation module and interface also has a *code signature*, generated from a secure hash of the bytecode making up the module, or the types specified in the interface, to enable references between modules and interfaces to be rapidly resolved.

The form taken by modules and interfaces will depend on the safe language being employed. In the case of Java, a module may map to a Java class; in Caml [Leroy97] a *structure* would be used for a module, and a *signature* would represent an interface.

5.2.4.1 Linking and Binding

Linking is the operation of introducing a new module into the system. In a system using hardware protection, there is typically no need for any verification by the system when a user loads and links new code. In a system using software protection, however, it is necessary that the system can verify that the module is safe to execute, using techniques such as those described in Section 5.2.3. Such verification will involve checking that all of the new module's imports can be resolved by existing modules, and that the code in the module respects the typesystem of the language and the types of the imported interfaces.

To access the elements of a module's interface, a session must be *bound* to that module. Binding to a module instantiates the module into the namespace of the requesting session. Sessions may bind to modules either by presenting the desired bytecode to the system (thus simultaneously performing the linking and binding steps), or by referring to the code signature of the desired module[4].

[3]It is important to bear in mind that just because the owner trusts the code *signers*, it does not mean that he trusts those whose code has been signed – the signature only serves to identify the author of the code, and does not verify that it is safe.

[4]When referring to a module by its code signature, the module must have been previously loaded into the system, possibly by a different session or at system initialisation.

Binding may be explicit – as a direct result of a user request – or it may be implicit, called recursively to satisfy the imports of another module that is currently being bound. During binding, the module's functions and data are initialised so that the interface exported by that module may be used by other modules in the same session.

5.2.4.2 Privilege

Some modules may have particular privilege levels associated with them. This may be the case if:

- the module allows unsafe access to the underlying Runtime,
- the module allows manipulation of important node state, or
- if the module contains proprietary algorithms that are not intended to be publicly accessible.

Such privilege may be expressed in various ways. An implementation of RCANE may represent privilege on a simple scale from "no privileges" to "full privileges", or more complicated capability-based models such as that proposed in [Hicks99b] may be used.

A module's privilege levels affect linking and binding:

Binding: When binding, the session requesting the bind must have sufficient privilege to bind to the desired module.

Linking: When linking, the session introducing the new module must have sufficient privilege to bind to all of the module's imports.

5.3 CPU Management

This section presents the abstractions used by RCANE to provide access to the CPU(s) on the node. The thread model was originally proposed in [Menage98b] and subsequently adopted in a modified form by the Active Networks NodeOS specification [Peterson00a].

RCANE's CPU management abstractions are structured so as to allow sessions to split their tasks between multiple scheduling classes, control the level of concurrency used for different sets of tasks, and to service those tasks efficiently. Three important abstractions employed are:

1. A *virtual processor*[5] (VP) represents a regular guaranteed allocation of CPU time, according to some scheduling policy (e.g. EDF [Liu73] or WFQ [Demers89]). All activities

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

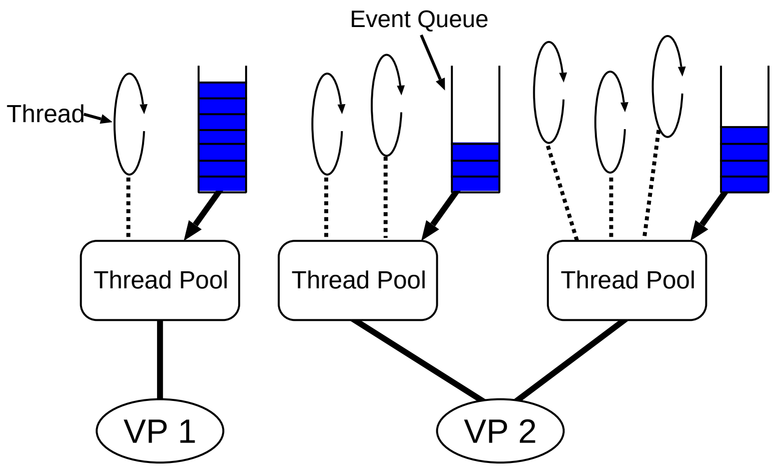


Figure 5.4: The RCANE CPU management architecture
Each VP represents a CPU allocation; thread pools permit events to be routed to worker threads.

carried out within a single VP share that VP's CPU guarantee. A session may have one or more VPs; by requesting multiple VP's, a session may organise its tasks into multiple independently-scheduled classes.

2. A *thread* is the basic unit of execution, and at any time may 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).

3. A *thread pool* is a collection of one or more threads. Each thread is a member of one pool. A thread pool acts as a queueing and dispatch point for packets and events. Each pool is associated with a particular VP; its threads are only eligible to run when its VP receives CPU time.

Figure 5.4 shows an example of the CPU management architecture used by RCANE. There are two VPs shown. VP 1 contains a single thread pool, with a single thread. CPU time reserved for VP 1 will be consumed by the single thread, processing events placed in the event queue of VP 1's thread pool (see Section 5.3.1). VP 2 contains two thread pools; these thread pools contain two and three threads respectively. The CPU time reserved for VP 2 will be shared between the threads contained in the VP's two thread pools.

The choice of scheduling algorithm used to select between the thread pools (and threads) within a single VP may be left to the application itself[6] – unlike the top-level scheduler used to select between VPs, there is no requirement that it be capable of supporting reservations or fine-grained scheduling. Since all the CPU time being consumed is accounted to the session, it

[6]The application may be restricted to choosing a scheduling algorithm from a set defined by the provider of the RCANE node and implemented in unsafe code within the Runtime. Alternatively, the node may employ extensible virtual machine [Harris99] or proof-carrying code [Necula97] techniques to permit an application to schedule its own threads in a safe manner.

may choose a simple scheduler that is efficient to run but which gives no particular guarantees to threads, or it may choose a more complex scheduler to satisfy some policy for sharing its CPU allocation between its different tasks.

5.3.1 Events

Each thread pool has an associated *event queue*. An event represents a callback into one of a session's functions. Events may be placed into the queue in two ways:

- An event may be requested to occur after a given delay. For example, a thread may specify that the (user-defined) `handleTimeout()` function be called with a given parameter in 100 milliseconds time. By default, an event requested in this way is sent to the thread pool in which the thread is running, but the user may specify that the event be sent to a specfied thread pool, or to the default thread pool for a specified VP belonging to the user's session.
- When a packet is received on one of a session's network channels (see Section 5.4), it is turned into an event that will invoke that channel's processing function – specified by the application – with the contents of the packet as an argument to the function.

Whenever a thread pool's event queue is non-empty (either due to newly arrived packets, or delayed events whose timeouts have passed) idle threads in the pool will be dispatched to process the events in the pool's queue. When a running thread has finished its task, it is dispatched to process the next event from the event queue if the queue is non-empty, or returns to the idle state if the pool's queue is empty.

This hybrid thread/event model allows sessions a great deal of flexibility in how they map their work onto threads. Traditional multi-threaded programming may be performed, at the expense of an increased amount of memory required for thread's stacks and greater effort on the part of the programmer to avoid concurrency bugs. Alternatively, by using the continuation-passing style of programming that is promoted by events, a single thread might be bound into a pool, serving multiple tasks consecutively from the same event queue, and thus reducing the amount of stack memory used, but with less concurrency. The level of concurrency allocated to a particular task is directly controlled by the number of threads attached to the task's thread pool.

This flexibility is extended to the level of CPU reservations. If a session has two tasks of the same importance to perform then a single CPU reservation (i.e. a single VP) may suffice. If however, a session has multiple tasks to perform, multiple VPs may be appropriate. For example, an application that is processing a network video stream, with packets arriving every 20ms and requiring 0.5ms to process, and which is also performing housekeeping processing that should not be permitted to interfere with the network processing, may choose to allocate two VPs – one with a 0.5ms slice of CPU time every 20ms of real time, for the network processing, and another with a less fine-grained guarantee for the periodic processing. A database server that wishes to provide services to multiple clients simultaneously may bind several threads into a pool to which incoming packets are directed.

5.4 Network I/O

Access to network flows is essential to allow mobile code to communicate both with its original source and with the other resources in the network with which it wishes to interact.

Rcane allows sessions to open *channels* to give access to network resources. A channel is a simplex or duplex flow of packets to and/or from the network.

Each channel that is capable of receiving packets has associated with it a demultiplexing specification (to select the incoming flow of packets to be directed to that channel) and a VP, which is used for RX protocol processing on that channel.

One of the significant sources of QoS crosstalk in a traditional operating system is the networking stack. In particular, the use of kernel threads to perform protocol processing can make it difficult to correctly account the resources consumed by a particular flow, and can lead to livelock situations[7] [Druschel96].

To prevent crosstalk between the network activity of different clients, all packets are demultiplexed to their receiving pools by the Runtime using a packet filter. Protocol processing is not performed on packets before demultiplexing. Each channel contains a FIFO queue of received packets – if this queue fills up, subsequent packets arriving for that channel will be discarded until such time as space becomes available in the queue.

At a later time, when the VP associated with the channel is eligible to receive CPU time, protocol processing occurs:

1. The packet is removed from the channel's packet queue.
2. Any required processing such as defragmentation or checksum validation is performed.
3. If specified by the application, further demultiplexing into sub-flows may be performed, based on the value of specified key fields within the packet data.
4. The contents of the packet are encapsulated in a callback event to the function specified for the particular flow or sub-flow. Incoming packets for each thread pool are stored in a per-pool packet queue. This is conceptually part of the per-pool event queue described in Section 5.3.1. However, in order that the timely delivery of events to a client is not compromised by a flood of incoming packets for that client, events in a pool's event queue have priority over packets in that pool's packet queue.

The time required for this protocol processing is entirely accounted to the VP associated with the channel, and thus to the session that opened the channel. Thus even a session that is receiving large amounts of network data will not be able to interfere with the level of resources received by other sessions. Moreover, if a session is receiving more data than it can effectively process, excess packets will be dropped at the bottom level of the system,

[7]Livelock occurs when little or no useful work is performed processing received data, as the system is too busy processing (and throwing away) more incoming data.

rather than consuming excessive buffer resources, or causing needless protocol processing to be performed before the packet is ultimately dropped. Ideally, such packet filtering and discarding will be performed in an intelligent network card [Pratt97, Pratt00], to avoid wasting any CPU cycles or bus bandwidth on data that is ultimately discarded.

An application may associate each flow or sub-flow with a particular thread pool. The callback event constructed by the protocol processing is placed in the packet queue for the specified thread pool. This allows different logical flows of packets to be processed by different thread pools, possibly running in different VPs to give better QoS isolation between the different flows.

This network architecture allows each session full control over decisions such as whether authentication is used – and if so, what kind – for packets on a given flow. For non-authenticated flows, a session can specify a function that 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. Other protocol layers such as reliability, ordering and fragmentation may similarly be composed as required.

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.

Figure 5.5 shows an example configuration of the Rcane network architecture. Sessions A and B have each opened a single channel to a network interface. Incoming network frames are demultiplexed in the network packet filter and placed in the appropriate channel FIFO (or dropped if that channel is full). These frames are not processed further until the client VP associated with that channel receives processing time.

In the case of Session A, no further demultiplexing is performed, and after protocol processing (which is accounted to session A's VP), the frames' payloads are placed on the packet queue for session A's thread pool. In the case of session B, the protocol demultiplexing delivers packets to one of two thread pools, each of which has its own CPU guarantee. Such a setup might be used when a client is performing two activities, both of which require large amounts of CPU time and one of which involves processing very few packets. For example, one activity might involve processing a network data stream, and the other may involve performing computations, directed by a low-bandwidth stream of control packets from the end-user. Rather than set up a separate channel for each activity, the client may decide that any crosstalk introduced by the additional multiplexing and demultiplexing of the few packets from the second activity into a single channel is small, and that the reservation costs of a second channel would be too great. Any crosstalk caused by this decision will be internal to the client, and will not affect other clients. Hence Rcane allows clients to make such

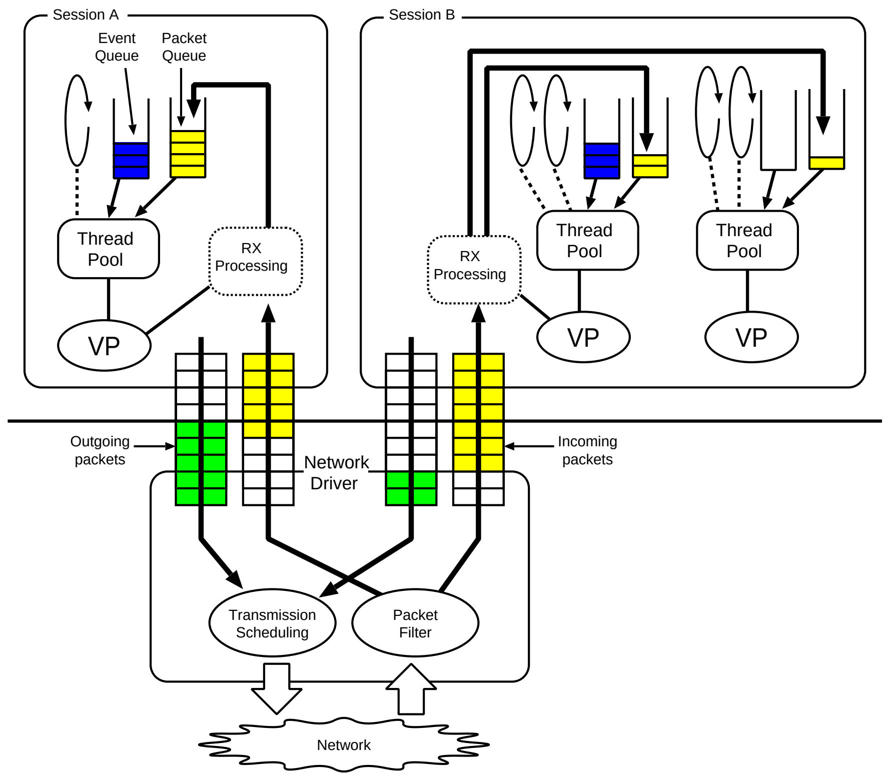


Figure 5.5: The Rcane network architecture

decisions based on their own criteria.

Packet transmission is similar to that for reception, with link-level frames being placed in an outgoing FIFO[8]. A transmit scheduler takes packets from the FIFOs according to the reservations made by clients, and sends them out over the network.

5.5 Memory

As described in Section 3.3.2, the memory managed by Rcane falls into five categories: network buffers, thread stacks, dynamically-loaded code, auxillary data structures, and heap memory.

Network buffers (discussed in Section 5.4) and thread stacks are accounted to the owning session in proportion to the memory consumed. The charges incurred for keeping code modules in memory are likely to be based on a system specific policy (e.g. it may be the case that linking to a commonly used module would be less expensive than loading a private module). Techniques for accounting for auxillary data structures were presented in Section 3.3.2.5.

[8]Protocol processing on the TX side is not shown in Figure 5.5 due to lack of space.

Heap memory presents more challenges. Safe languages that permit dynamic allocation of memory have to ensure that pointers cannot be used after the memory to which they refer has been freed – such a pointer is known as a *dangling pointer*. Since the memory referenced by a dangling pointer may have been reused for a different type of object, it represents a potential violation of the safe typesystem. To avoid dangling pointers, a garbage collector [Wilson92] (GC) may be employed to ensure that no memory block is freed while pointers to it remain in existence.

The heap memory architecture employed by RCANE must therefore provide a garbage collector; in order to meet the aims of isolation and accountability, it should support the following features:

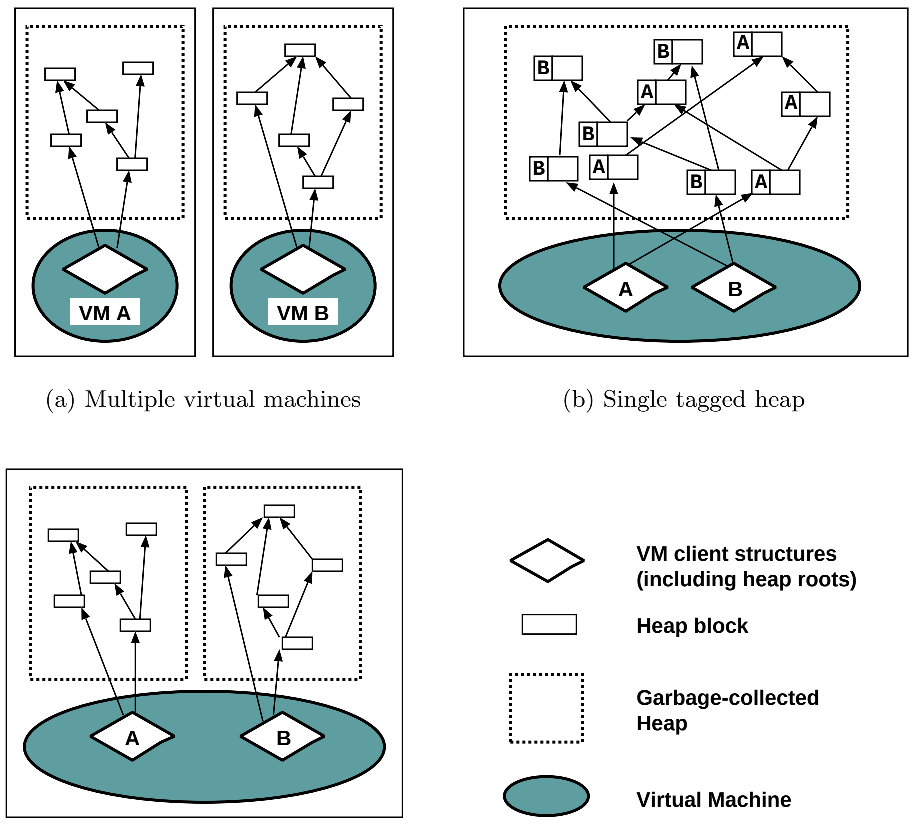
- Efficient tracking of the memory usage of each session.
- Prevention of crosstalk between sessions due to GC activity.
- Ability to revoke references from other sessions when a session is deleted.

Three different possible approaches were considered, shown in Figure 5.6 and described in the following sections.

5.5.1 Multiple Virtual Machines

The traditional approach to providing isolation between multiple applications is to run each application in a separate virtual machine in a separate OS process. Most virtual machines assume that only one *user* is using the machine, even if parts of the code being executed are not trusted to have full access to the system. By allowing each VM to run independently, we preserve the validity of this assumption. This provides a good level of isolation, since each application's access to resources may be controlled by the OS; however, using multiple VMs has certain drawbacks:

- Communication between different applications is made considerably more difficult.
- Sharing of VM data structures and code is not straightforward; for traditional native applications, code sharing is effected by sharing text segments when they are loaded from disk. In an environment such as RCANE, where code is supplied over the network, and potentially transformed into native-code using JIT techniques, more complicated strategies would be required, involving co-operation between VMs to identify and coalesce shared code pages.
- It is less scalable due to the overhead associated with a new process (both in terms of cycles required to create the process, and system and user memory required for the process).



(a) Multiple virtual machines

(b) Single tagged heap

(c) Multiple heaps

Figure 5.6: Approaches to providing memory accounting

5.5.2 Single Tagged Heap

This approach uses a single garbage collected heap (and a single virtual machine) for all applications. Each memory block is tagged to indicate which application allocated that block. Whenever a block is allocated, the size of the block is added to the tally for that application. When a block is garbage collected, the owning application's tally is decremented. Use of a single heap may reduce the amount of wasted empty space, by permitting the application of statistical multiplexing techniques to reduce the total amount of memory required on the node.

There are five significant drawbacks to the single tagged heap approach:

1. If pointers are permitted to be passed between different applications, then one application may retain a pointer to objects allocated by another application. This may result in blocks of memory that are no longer held by the allocating application (and thus ought to be garbage collected and decremented from the application's tally) remaining live. Thus the application is no longer in charge of its own resources.

Furthermore, if the allocating application exits or runs out of resources, there is no way to free the memory that it allocated, without creating a dangling (unsafe) pointer. This could lead to the situation where a series of malicious clients each create a session to allocate large amounts of memory and make references to this memory available to other (possibly innocent) sessions. When these sessions exit (and hence are no longer charged for resources) it may be impossible to free the memory thus allocated.

One possible solution to this problem would be to specify that sessions are held (jointly) responsible for any memory that can be reached from their heap roots. However, performing such analysis is expensive; moreover in the scenario presented above, the result would be that unwary clients could end up footing the bill for the storage allocated by other sessions and stored in places accessible to them.

2. Garbage collection incurs a processing overhead, in order to locate unreachable memory objects and return them to the free memory pool. With a single heap, there is no straightforward way to account the CPU time required for garbage collection activity to applications in proportion to the amount of activity for which they were responsible.

3. Even when other applications do not retain pointers to memory blocks owned by an application that exits, it is still not possible to efficiently free the application's memory. Since all memory blocks are mixed together in the heap, with no partitioning on a per-application basis, it is necessary to trace through the object graph of the exiting application in order to identify and free its memory. Thus, unless CPU time is devoted to such a task immediately when an application exits, the blocks owned by an application persist as garbage for some period of time after that application's session is destroyed.

4. If one application causes significant heap fragmentation (either through malice or through lack of knowledge of the node implementation's heap layout) then the useful memory available for all applications may be severely reduced[9].

5. There is an overhead in the form of the application tag on the front of each memory block. In the case of a language such as ML [Milner97] that makes heavy use of small immutable memory blocks, such tagging would add substantial overhead to the total heap memory usage.

The JRes project [Czajkowski98] made use of a single tagged heap. JRes used binary-rewriting techniques to cause all memory allocations to also call a routine to increment the current thread's allocation total; object destructors were similarly rewritten to decrement the thread's allocation. Apart from problems due to the example implementation being written mostly in Java with little support from the JVM[10], associating allocations strictly with a thread can cause problems if a user wishes to have multiple threads co-operate.

[9]This drawback applies only to Mark/Sweep garbage collectors – using a Stop/Copy collector would remove the problems associated with memory fragmentation.

[10]Byzantine behaviour in destructors could cause the memory allocations to be decremented before objects were actually garbage collected, thus allowing users to consume more memory than was actually being accounted to them.

5.5.3 Multiple Independent Heaps

A hybrid approach is to make use of a separate heap for each application, within a single virtual machine. This facilitates accounting for each application's usage of memory, since it is only necessary to track the amount of memory reserved for that application's heap, regardless of whether that memory is free, live, or garbage.

It also makes it possible to account the CPU time required for garbage collection to each application, since garbage collection can be scheduled along with ordinary execution for each RCANE session.

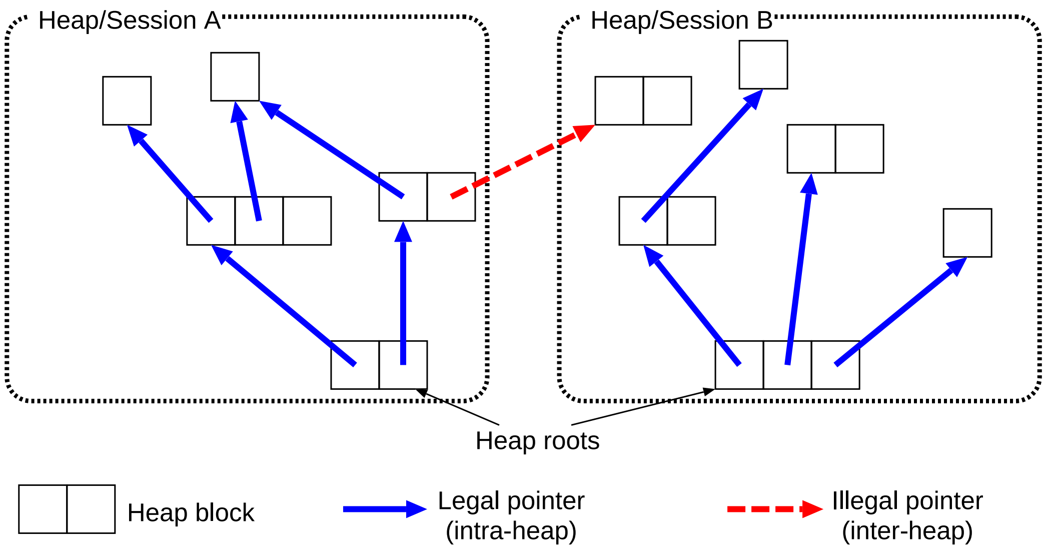


Figure 5.7: Dangling pointers caused by inter-heap references

When using multiple independent heaps, it is essential that references between objects in different heaps be prevented. If such inter-heap references were to exist, the type-safety guarantees provided by the use of a safe language could be broken, as demonstrated in Figure 5.7: the heap belonging to session A contains a reference to an otherwise unreferenced object in session B's heap. Since the garbage collectors are independent, B's collector will consider the referenced object to be garbage, and return it to the freelist. This will leave a dangling pointer from A's heap; if this location is later reused for a different object, the typesystem will be violated and arbitrary behaviour may occur.

Such inter-heap references may be prevented either statically (by preventing access through the type-safety of the language) or dynamically (at runtime). Dynamic prevention requires that every time a reference is written to a field of a heap object, a check is performed to ensure that the heap of the source and destination objects is the same. The Conversant project [Bernadat98] implemented a dynamic checking system in Java, and experienced a 15% overhead on writes when using interpreted code, and a 55% overhead when using JIT-compiled code. KaffeOS [Back00] used similar checks, adding 25–41 cycles to the cost of a write (excluding cache effects).

In a type-safe language pointers may not be generated arbitrarily, and may only be obtained by allocating an object on the heap or reading an existing pointer from a variable. Thus if each session begins with pointers only to objects in its own heap, and the Runtime does not permit pointers to be passed unsafely between heaps, there is no way for a malicious programmer to generate an inter-heap reference – he can only allocate a new object (in the local heap) or access a pointer already stored in a heap object (which is guaranteed to be a reference to an object in the local heap). Thus the property that no inter-heap references exist is maintained, and requires no overhead for runtime checking. However, it does make communication between different sessions more expensive, since values must be copied by Rcane from one heap to another.

A second reason for choosing static rather than dynamic inter-heap reference prevention is due to the overhead associated with root tracing. When garbage collection occurs for a given heap, an important part of the collection activity involves tracing all the roots[11] of the heap to provide a initial set of live heap objects. If static inter-heap reference prevention were not employed, it would be possible for threads executing in one session to have pointers in their *stacks* which referred to objects in other heaps. Thus, all stacks in the virtual machine would need to be traced to check for roots when garbage collecting any heap. This is an unsatisfactory solution for two reasons:

- The time (accounted to the client owning the heap being collected) required for the root tracing part of garbage collection would increase with the number of threads in the entire system, reducing scalability and contributing to crosstalk between sessions – the amount of GC work done by a session would be dependent on the number of threads created by other sessions.
- When performing root tracing on a thread's stack, it is necessary to suspend execution of that thread. This is because write barriers[12] are (usually) not used on stack locations due to the substantial overheads that would be incurred on every write to a stack location.

  Thus, dynamic reference prevention would result in all threads on the system being suspended due to garbage collection activity in other threads. This would cause substantial jitter in the CPU QoS received by sessions, reducing the isolation between supposedly independent sessions.

Conversant [Bernadat98] provided a separate heap for each user application. Static members of classes in the system heap were accessible – when permitted through standard Java abstraction mechanisms – to untrusted code; this required that garbage collection of the system heap caused root tracing (and hence suspension) of all threads in the system, since any of them might have pointers to system heap objects on their stacks. Furthermore, Conversant tagged every object with the identifier of its heap, leading to increased storage usage. The J-Kernel [Hawblitzel98] simulated multiple heaps in Java by requiring all accesses between

[11] *Roots* are non-heap objects such as static variables, registers, and stack locations, which may contain references to heap objects.

[12] A *write barrier* is a synchronisation protocol between the garbage collector and the mutator (heap user) to ensure that concurrent garbage collection is performed correctly. For more details see [Wilson92].

different clients' data structures and objects to be through special *capability* objects. Invocations on these objects caused parameters to be marshalled from the client domain to the server domain in a similar way to RCANE's service calls described in Section 5.6; however, all threads in the J-Kernel execute in a single heap, so while some desirable properties of heap separation – such as the ability to revoke services and terminate sessions – are achieved, other properties such as accounting for memory usage and preventing GC crosstalk are not straightforwardly realisable.

5.5.4 RCANE memory architecture

The RCANE memory architecture is based around the single virtual machine, multiple heaps solution. Each session is given its own independent heap. The maximum reserved size for this heap may be configured by the session, by requesting a particular size from the Core.

An incremental garbage collector – which processes a small portion of the heap each time it is invoked, rather than processing the entire heap in a single invocation – is employed to prevent excessive interruptions to execution. Such a property is essential to prevent the unpredictable nature of garbage collection from causing clients to miss their deadlines. Each session may tune the parameters of GC activity – such as frequency and duration of collection slices – in order to trade off responsiveness against overhead.

Charging is 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 and more accurately reflecting the load placed on the system's memory by each client.

Associated with each heap is a set of threads that may have references in their stacks to objects within that heap. The threads in this set are those that must be suspended in order to perform root-tracing on the heap. Conversely, associated with each thread is a set of heaps to which it has access. A thread gains access to a heap when it invokes a *service* (see Section 5.6) and loses access to that heap when it returns from the service invocation. Thus, the set of threads associated with a session's heap is generally a (non-strict) superset of the set of threads created by that session.

5.6 Inter-Session Services

As described in the previous section, in the RCANE architecture each session executes in its own private heap. Thus pointers between different clients' sessions are not permitted, with the effect that communication between code executing in different sessions through direct procedure call is not possible.

In many cases this does not present a problem; since RCANE aims to avoid shared servers on the data path, and instead to execute data-path code (such as network protocol processing code) in the context of the session which is requesting such activity, such isolation is actually desirable as it simplifies the provision of guaranteed access to node resources.

In some situations, however, it may be necessary or desirable to communicate with applications or services running in 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).
- As described in Section 5.2.1.2, EEs in Rcane are treated as libraries to be instantiated by a client's session. Some EEs may have a requirement to maintain state shared between multiple instances of an EE on a node, implying the need for communication between sessions.
- Some sessions may wish to export services to other sessions running on the node (e.g. extended network routing tables, or access to proprietary data or algorithms).

In each of these cases, code executing within one session requires a way of interacting with code or state within a different session. Such interaction could be implemented by a client session with no extra intervention on the part of Rcane by sending network packets through a loopback network interface to another session.

The principal drawback of such an approach is inefficiency. A secondary drawback is the loss of type-safety.

A more efficient and type-safe solution may be provided with support from the Rcane Runtime layer. Such support can draw on previous research into Remote Procedure Calls (RPC), traditionally between two processes executing in different hardware protection domains (possibly on distinct computers).

5.6.1 Types of RPC

RPC facilities may be provided in two main ways: *message passing* and *thread tunnelling.* In [Lauer79] it was proposed that these two methods are duals of each other[13], and that they can be made equivalent in performance terms *provided that the underlying primitive system operations are equally efficient in each case.* However, the QoS guarantee requirements of Rcane, combined with the lightweight context switch available through use of a safe language, mean that the two styles of RPC communication are not necessarily equally applicable. The following sections examine both message passing and thread tunnelling in the context of the Rcane architecture.

5.6.1.1 Message Passing

RPC between two separated computers, (as presented originally in [Birrell84]) requires message passing. The sequence of events which occur during a message passing RPC is as follows:

[13]In fact the argument in [Lauer79] is expressed in terms of *monitors* ([Hoare73]) rather than thread-tunnelling – however, a similar duality can be drawn between a thread-tunnelling system and a multi-threaded message passing system.

1. The calling thread of execution in the client process marshals a procedure reference and a set of arguments into a buffer, transfers this buffer in some way to the server process, and blocks awaiting a reply.

2. A thread within the server process reads the procedure reference and the arguments out of the buffer, and constructs a call frame to invoke the specified procedure with the supplied arguments. On return from the invocation the server thread marshals any results or exceptions into a reply buffer and transfers it back to the client.

3. The client thread is unblocked, unmarshals the results, and returns to its caller or raises an exception as appropriate.

Many traditional RPC systems – both inter- and intra-machine – use message passing [Birrell84, Birrell93, Roscoe95].

5.6.1.2 Thread Tunnelling

The time taken to reschedule to the server, and then back to the client, can add significant latency to the time taken for a message passing RPC to complete. Furthermore, the server may not be eligible to receive CPU time at the point of the invocation, increasing the latency further. Thread tunnelling[14] [Wilkes79, Bershad90, Chase93] aims to reduce the latency and synchronisation costs associated with message passing on a intra-machine call by switching the calling thread into the protection context of the server. This avoids the overhead of calling into the scheduler and waking the server thread. If the arguments are small (i.e. can fit within processor registers) it may be possible to leave the arguments within the processor registers and avoid the marshalling step entirely, thus reducing the latency further.

Since the thread owned by the client is executing the server's code, it is necessary to require that the server not be permitted to perform certain operations on the thread, such as terminating it. This can ensured by preventing the server from obtaining a handle on the current thread ([Hawblitzel98]), or by wrapping each service call segment in a conceptually separate thread ([Ford94]).

5.6.1.3 Discussion

In [Roscoe95], it is argued that thread tunnelling is unsuitable for systems seeking to provide resource isolation for the following reasons.

Increased crosstalk: If a thread has tunnelled into a server domain, it is no longer possible for the client to be responsible for scheduling that thread (i.e. the system kernel must be responsible for scheduling not only processes, but also threads within processes), reducing the control that a process has over the scheduling of its threads. This in turn may lead to

[14]Thread tunnelling may also be referred to as *thread migration*.

increased *crosstalk* should a process need to schedule a thread according to an application defined policy in order to meet a deadline.

In a system such as the original implementation of Nemesis, where each process has only a single CPU allocation, this criticism is valid; however, the architecture used by RCANE permits each client to reserve multiple guaranteed allocations of CPU time, and makes it straightforward for clients to assign different activities to different CPU allocations.

Furthermore, there is a significant source of crosstalk in a message-passing system that is absent or attenuated in a thread-tunnelling system.

In order to provide useful QoS guarantees to its clients, each server must schedule the CPU cycles consumed by its threads according to the priorities of the clients that are calling it. This complicates the server substantially. [Ingram99] describes the implementation of an X server to perform such scheduling under a soft-real-time Unix system. Previous work on the Nemesis display system (described in [Barham96] and extended by the author) found that a restricted form of thread tunnelling provided access to the framebuffer with both lower latency and lower crosstalk compared to the earlier message-passing display server, without substantially affecting the ability of applications to control their own scheduling policies. The thread tunnelling mechanisms permit each client to be accurately charged for the CPU cycles[15] consumed on its behalf in the server.

Loss of control: Separately from issues of scheduling, since the client is executing the server's code, the server can control when the service call returns, and may (due to malice or programming errors) fail to return to the client. In this situation, the client sees the thread as effectively blocked, and thus cannot continue until the call returns. [Roscoe95] argues that when using message passing, the client still has full control over its own threads, and may continue with other processing, or decide to cancel the service call and raise an exception to the caller. In a properly event-driven and parallelised client, message passing does allow the client to be relatively unaffected by a server failing to return. Experience shows, however, that many uses of message passing RPC treat it as a synchronous call, and hence will block until the server does return. Thus, provided that the thread-tunnelling mechanism provides a way for the client to abort in the event of a call taking an excessively long time, message passing and thread-tunnelling are little different in this respect.

Message passing can be pipelined: [Roscoe95] suggests that message-passing can amortise the overhead of a single context switch over several RPC invocations, effectively pipelining requests. This approach has been shown ([Black94]) to be effective when the communication being performed is in the form of bulk data transfer from client to server or vice versa, in which case clients have no need to block after each request awaiting a reply. However, for traditional RPC, generating sufficient outstanding invocations in the client to benefit from such amortisation would require that the client have a very large amount of inherent parallelism available in its work to be exploited.

[15] Note that if the server is being used to multiplex access to a hardware device or other shared resource where coarse-grained synchronisation may be required, such scheduling may still be necessary.

A further problem with message passing is the availability of server threads. When a message passing RPC invocation is received by a server, it must be processed by a server thread.

This thread may be obtained in one of several ways, each of which has its own disadvantages:

- The server may maintain a fixed set of worker threads to service all invocations on an RPC binding (or set of bindings). Each thread services a complete invocation before starting to service the next invocation (or blocking if no more work is available). In this case, any invocation arriving when all worker threads are busy will wait until a thread is free.

  There are two main disadvantages with this approach. The first is that it becomes an obvious source of crosstalk between applications – unless the server pre-allocates sufficient worker threads (most of which are likely to be sitting idle for most of the time) clients with short deadlines can block waiting for invocations on behalf of under-resourced clients to complete. The second is that in the presence of recursive RPC between several domains, it is possible to enter a deadlock situation[16].

- The server maintains a set of worker threads (possibly just a single thread) and a set of current invocations in various states of progress[17]. Each thread repeatedly requests an outstanding invocation from the server's scheduler and performs a small amount of work upon it.

  Such an approach may be usefully employed in special purpose servers – such as event-driven web servers – where the likely properties of the request are well-known in advance and switching a thread between servicing different requests may be performed fairly straightforwardly. This approach is not, however, particularly suited to the provision of a general purpose RPC server; it is necessary to structure each individual service in order to break requests up into small chunks of work, each of which may be served in a short amount of time without blocking. A further consideration is that the server must employ some form of scheduler to decide which invocation each thread should be working on at any given time; the disadvantages of this are discussed above, on page 77.

- The server may fork new worker threads to service incoming requests (possibly maintaining a cache of pre-forked idle threads up to some specified limit). This approach is employed by many message-passing RPC systems, including [Birrell93]. It avoids the crosstalk and deadlock problems associated with a fixed set of worker threads. However, it incurs significant additional overheads due to the costs of creating a new thread; such costs may also themselves introduce additional crosstalk.

These drawbacks are removed by the use of thread-tunnelling. The thread is supplied by the client, so no overhead is incurred in creating a new thread. The client has more control over the level of concurrency used for its operations within the server, and no less control over the scheduling than if the thread was executing within the client's own session.

[16]Such deadlocks have been regularly observed by the author within the system services in Nemesis.

[17]States might include `Arrived`, `Unmarshalled`, `Returned`, `Marshalled`, `Completed`, as well as service-specific states.

For the reasons given above, RCANE employs thread tunnelling to provide inter-session services. It should be noted that while RCANE and Nemesis both seek to provide QoS in their RPC services, a major architectural difference between them is that clients within RCANE are separated through software protection, whereas Nemesis uses hardware protection. This results in the thread-tunnelling approach being somewhat more practical in RCANE than in Nemesis.

5.6.2 Priority Inversion

Notwithstanding the arguments presented in the previous sections, thread-tunnelling is not without potential pitfalls. One of the most obvious is that of priority inversion, identified in [Lampson80]. This can occur when a client thread with limited resources is executing within a mutual exclusion region.

In a traditional system using priorities rather than guarantees, priority inversion may lead to long-term starvation – if a high priority thread wishes to enter the mutual exclusion region, it must sleep until the low priority thread leaves the region; in the meantime a medium priority thread may preempt the low priority thread preventing the high priority thread from making further progress. In a system providing guarantees, long-term starvation due to priority inversion is less likely since even a low-priority task can be guaranteed a certain level of access to CPU resources. However, priority inversion can still create situations in which tasks with fine-grained guarantees are held up due to an under-resourced low-priority task spending a long period of time within a mutual exclusion region[18]. This situation should not arise to the same degree in a message-passing server – provided that the server itself has sufficient resources guaranteed to it that it is not starved by other applications – but it should be noted that any system that combines the provision of resource guarantees with mutually exclusive access to shared resources will suffer from similar problems.

[Lampson80] proposes *priority inheritance* as a solution to the problem of priority inversion: whenever a thread holds the lock on a mutual exclusion region, it runs with the highest priority of any thread waiting to enter the region. A similar approach is proposed in [Waldspurger94] using the transfer of *lottery scheduling* tickets. This solution is less applicable to a system using resource guarantees rather than priorities, since it could result in one client being charged for resources consumed by the server code whilst doing work done on behalf of another client. Two possible alternatives are:

- Allow the server to "underwrite" a thread within a critical section with a resource guarantee of its own.

  This will enable the server to ensure that whilst in the critical section, the client thread receives at least a given level of resources; thus the server may bound the crosstalk caused by the mutual exclusion on the critical region.

- Permit the server to detect a client with a small resource allocation, and either (a) raise an exception to the client, or (b) block the client thread until the scheduler can guarantee

[18]In the case of a malicious (rather than simply resource poor) client, such situations may even be deliberately engineered.

that it will be able to completely execute the critical region without preemption. Such a mechanism is used in *Rialto* [Jones97] to provide dynamic real-time guarantees.

This solution potentially results in less overhead within the scheduler; however, it can result in under-resourced clients never gaining access to the critical region.

Rcane employs the first of these solutions to limit crosstalk in inter-session communication, due to its straightforward and wide-ranging applicability.

5.6.3 Recovery from Client Failure

A second potential problem with thread-tunnelling occurs due to the fact that sessions running under Rcane are charged for the resources that they consume, and hence a session (and its threads) may be destroyed if it exhausts its resources.

If a server exits or is destroyed whilst a client is executing in a service within that server, an exception may be raised in the client's domain at the point where control passed from the client to the server.

If a client exits or is destroyed whilst executing within a server, the situation is more complicated. Since the server's code is being executed on the client's thread, and that thread is being destroyed, the server must be given a way to recover gracefully from the destruction of the thread. If the server is manipulating shared state at this point, simply destroying the client thread may leave the server's data in an inconsistent state. Various strategies to allow such recovery are possible:

Uninterruptible sections: PERC [Nilsen98] defines extensions to Java that allow a block to be marked as `atomic` – such a block is guaranteed not to be preempted by the scheduler or asynchronous exceptions. However, `atomic` sections are restricted in the type of code that they may contain, and their execution time must be statically analysable and below a certain threshold. Such an approach may not be practical for servers that make non-trivial manipulations of shared state.

Rollback events: Since Rcane uses a garbage collected heap for each session, it is not necessary for the server to explicitly free any memory allocated whilst servicing the client. However, it *is* necessary for the server to be able to roll back any changes being made to shared data structures.

A possible solution would be to permit the server to specify a *rollback event* at the point of entering a critical region whilst servicing a client call. The rollback event encapsulates the actions that the server must perform in order to restore the invariant of the critical region. Should it happen that the client thread is destroyed whilst still in the critical region, the rollback event is posted to the thread pool specified by the server (with the critical region lock still held); the server's thread then executes the recovery actions and releases the critical region lock.

The use of rollback events is applicable to the most simple critical sections, where it is possible to wrap up a set of actions to restore the critical section's invariant. It is less suitable for more complex critical sections, particularly if interactions with other sessions or the network, etc., occur during the section.

Transaction logging: The use of *transactions* [Gray93] in database systems and distributed systems to provide the ACID[19] properties has been heavily researched. Each set of operations on shared state is packaged up into a transaction. Upon completion of the transaction, the operations are *committed* or *aborted* as a single group; if the transaction is aborted, then its effects are undone and will never be seen by other concurrent processes[20].

In [Seltzer96], the use of transactions was proposed for untrusted kernel extensions – called *grafts* – in the VINO operating system. Grafts are executed safely through the use of SFI [Wahbe93]. All accesses from grafts to regular kernel state are through *accessor functions*. The accessor functions both implement policy on access to kernel state, and push *undo functions* on to a stack associated with the transaction. If the transaction is aborted, these undo functions are called to reverse the effects of the transaction.

The use of a transaction mechanism for dealing with the unexpected abort of clients is similar semantically and in complexity to the use of rollback events, although it shifts the burden of implementing the rollback mechanism from the servers on to any services that those servers might invoke – thus any server that *might* be called as part of a transaction would be required to implement undo functions.

Backup threads: An alternative possibility is to allow the server to specify a "backup thread" when entering the critical region. In the event that a client thread is destroyed within the critical region, the state of that thread may be transferred to the backup thread, which continues from the point where the client thread was destroyed.

This approach has several advantages over the use of rollback events and transactions:

- It removes from the server the potentially complicated requirement of structuring its critical regions such that the region's invariant can always be restored using a rollback event.
- It is likely to be cheaper in the common case. The overhead of assigning a thread to be used as a backup is likely to be a small and constant cost compared with the overhead of building a rollback event or transaction – since the vast majority of service invocations are expected to complete successfully, this suggests that backup threads will typically be more efficient. Also, the action of specifying a backup to be used in the event of failure may be optimised in the Runtime, whereas allocating a rollback event will need to be server specific.
- Although crosstalk may be caused if no backup thread is available when entering a critical region, the server may bound the level of such crosstalk through prudent allocation of sets of backup threads to sets of critical regions. Furthermore, the fact that a critical region is already an inherent source of crosstalk suggests that any additional crosstalk effects may not be severe, and should be no worse than that experienced on a transaction-based system.
- It may be integrated with the resource backup mechanism described in Section 5.6.2.

[19]Atomicity, Consistency, Isolation and Durability.

[20]Weaker forms of transaction semantics are possible, and may be appropriate in some circumstances.

For the reasons given above, RCANE uses backup threads to support recovery from client failure in inter-session services.

5.6.4 RCANE Services

As discussed in Section 5.6.1, RCANE uses a form of thread-tunnelling between different heaps to provide efficient and flexible inter-session services. The remainder of this section presents further details of the RCANE inter-session service architecture.

Services exported from one session to another are represented as closures (functions with associated state). To a client or a server, the service invocation appears to be a normal function invocation. However, since the client and server are using different heaps, the Runtime is required to provide support for transferring parameters from the client to the server, and copying results back from the server to the client.

As in many traditional RPC systems, a single service involves both client-side and server-side data structures. The client-side *handle* contains information required for marshalling between the two heaps, and a reference to the server-side *exporter*. The exporter contains a reference to the real function that implements the service and a table of handles that other sessions hold for this service.

A service invocation normally involves the following steps in the Runtime:

1. Check that the service reference in the handle has not been revoked (see Section 5.6.5).
2. Add the current thread to the list of threads that have access to the server's heap.
3. Copy parameters to the service invocation into the server's heap. Since the Runtime has direct access to both heaps, there is no need to marshal the parameters into a buffer and then out into a data structure in the server's heap – this copy may be performed directly. This step may be omitted if the parameters are small enough to fit within registers.
4. Make the server's heap the primary heap of the current thread. This ensures that memory allocations made during the course of the service invocation use the server's heap.
5. Invoke the underlying function.
6. Make the client's heap the primary heap of the current thread.
7. Copy the results from the server's heap to the client's, as in step 3.
8. Remove the current thread from the list of threads with access to the server's heap.
9. Return the results to the client.

Any work carried out by the server during the invocation is performed using the client's thread, and thus accounted to the client's CPU allocation. Figure 5.8 shows the `Service` interface provided for creation and manipulation of services.

```
type α → β service

type α → β handle

type permissions = Private | Sharable

type memlimit = int

type copyInfo = Tree | Graph

type reclaimInfo = Retained | Collected

type serviceAttr = permissions * memlimit * copyInfo * reclaimInfo

exception Revoked

exception Aborted

(α → β service * α → β) create (func : α → β; attr : serviceAttr);

β invoke (s : α → β handle; parms : α; timeout : time);

void destroy (s : α → β service);

void release (s : α → β handle);

(α → β handle) get_handle (wrapper : α → β);
```

Figure 5.8: The `Service` interface

- `create()` takes a "normal" function and returns a service descriptor and a service function – invoking a handle referencing the returned service will cause the session switch described above. Thus invoking a service appears to the client to be the same as invoking an ordinary function.

  Other parameters to `create()` specify:

  - Whether the service may be passed around between different clients, or may only be passed between the server and clients.
  - The maximum amount of data to be copied when invoking the service.
  - Whether the parameters to the function are in the form of a tree or a graph (allowing optimisations to be made when copying).
  - Whether the service reference is *weak* or *strong*. A weakly referenced service will be garbage collected (and revoked) once no pointers to it remain in the server's

session, whereas a strongly referenced service will persist while one or more clients retain handles on it.

The memory limit passed to the `create()` function allows servers to prevent DoS attacks by clients. Without such a limit, a client could e.g. pass in a very long list as a service argument and cause excessive amounts of the server's memory to be allocated. 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.

- `invoke()` makes a service invocation as described above. `invoke()` is not generally needed, since when a client makes a normal invocation on a service the Runtime will cause the service invocation to take place. However, `invoke()` gives the client greater control over the execution of the service invocation than is possible through the normal closure interface. The current specification allows the client to specify a timeout, after which period the Runtime will abort the service invocation and drop back into the client. This allows a client to prevent a server from stealing its threads. Other client-side control over the invocation could be added to future versions of the interface.

- `destroy()` allows a server to withdraw the service controlled by the given service descriptor. Clients attempting to invoke the service in future will experience a `Revoked` exception.

- `release()` allows a client to release a handle on the given service.

- `get_handle()` allows a client to obtain the service handle from a service closure, in order to call `invoke()` or `release()`.

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. The copy-by-reference nature of services also allows large data structures to be split in a natural way using services of type `unit` $\rightarrow \alpha$, to avoid excessive data copying.

The ability to pass services between sessions is roughly equivalent to the capability passing used by the J-Kernel [Hawblitzel98] to pass object references between threads in different protection domains. J-Kernel protection domains consist of disjoint set of objects in a single heap, achieving similar functional isolation to RCANE but without the QoS isolation provided by independent garbage collectors.

Service passing is also similar in semantics to – although substantially more lightweight than – the interface passing performed by RPC systems such as CORBA [OMG98], Java RMI [Sun98] and Network Objects [Birrell93]. It differs from the explicit binding architecture used by the standard Nemesis IPC implementations and traditional RPC systems such as SUN RPC [SUN88] in that services are implicitly bound to when clients import them; however, the `Service.get_handle()` operation allows a client to access the underlying handle of a service, permitting explicit control if so desired.

5.6.5 Abnormal Termination

It is expected that almost all Rcane service invocations will finish normally, by returning a result to the client. However, there are certain exceptional conditions that Rcane must handle should they occur:

Uncaught server exception: If an exception is thrown by the code executing in the server, and this exception is propagated out of the service invocation without being handled, the exception must be trapped by the Runtime. The Runtime is responsible for copying the exception (along with its parameters, if any) back to the client's heap. Such copying occurs in the same way as with normal service results. Once the exception has been copied to the client's heap, it may safely be thrown by the Runtime (in the client's session) to the client's exception handling code.

Server revokes service: It is important that servers are able to revoke services that they have previously offered to clients. To this end, Rcane provides the `Service.destroy()` function, which a server may call on any service that it has created.

`Service.destroy()` marks all client handles for a given service as revoked, ensuring that further invocations by those clients will fail with a `Revoked` exception. Thus, when starting a service invocation, Rcane must check whether the client's handle still contains a valid service descriptor; if the descriptor is no longer valid, a `Revoked` exception must be raised to the client. Calls that are in progress at the time that the service is destroyed are aborted and the calling threads are returned to their originating sessions.

Aborted invocation: Rcane allows clients to abort invocations (perhaps due to a timeout, or due to no longer requiring the results of an invocation). If such an abort occurs, processing within the server invocation must halt, and the thread of control returns to the client. Execution continues with an `Aborted` exception being thrown, to inform the client that this invocation was aborted.

If at the time the abort occurred, the server was executing in a critical section and had registered a backup thread (see Section 5.6.2) the execution state of the client thread must be transfered to the server's backup thread. The backup thread may then be used to allow the critical region to complete safely. Once the critical region has completed, the backup thread may be released.

Client thread destroyed: The client thread may be destroyed during the course of the service invocation – possibly due to the client calling `Session.exit()`, or the client's session being terminated due to exhaustion of its resources.

In this situation, the behaviour on the server side is similar to that when the invocation is aborted – from the server's point of view it is irrelevant whether the client aborted the invocation or was destroyed. Behaviour differs on the client side – since the client thread no longer exists, it is not necessary to raise an `Aborted` exception; rather, the resources associated with the client thread may be simply released or recycled.

Server exits: In a similar fashion to the previous case, the server may be destroyed during the course of the service invocation. If this occurs it is equivalent, from the client's point of view, to the server revoking the service – thus a `Revoked` exception is raised to the client. Since the server resources are being destroyed, there is no requirement to transfer thread state to backup threads that may have been registered by the server; these threads, along with the shared state that such threads protect, will be destroyed at the same time.

In each of these cases, Rcane must ensure that the integrity of the system is not compromised by the abnormal termination; furthermore, since such terminations are expected to be rare compared to successful invocations, they should not adversely affect the performance of successful invocations more than is necessary.

5.6.6 Service Modules

In order to pass interface references as parameters or results of an RPC call, it is necessary to be able to "bootstrap" a client by providing a handle on an initial service or set of services. From these initial services, the client may obtain further references. This bootstrap may be performed in various ways:

- Using a trader with a well-known address, and allowing clients to lookup services by name. This method is used by Modula-3 Network Objects [Birrell93] and Nemesis.
- Using a shared filesystem to store "stringified" references to services, which may then be retrieved by clients. This method is used by many CORBA implementations, such as OmniORB [ORL97][21].
- Providing a system-generated handle to a bootstrap service. This method is more applicable to intra-machine services than to a fully distributed system. Nemesis provides newly created processes with a pre-generated connection to the system *Binder* (a service which is used to create further connections to other servers).

Rcane utilises the module space to provide a bootstrapping method that is similar to a trader. Some modules may be specified at load time as *Service Modules.* In general, a module exports a collection of functions in its interface. In the case of a service module, all of these exported functions must themselves be services. When a client in a different session links against the service module, handles for the services exported by the module are created in the client session. Invoking an operation on one of these handles causes a service invocation.

A service module may have either *shared state* or *per-client state.* A shared service module is initialised once, when it is created by the server. When clients link against it, the handles that each client receives all refer to the same set of services. Thus it is suitable for services that have no particular security requirement on them, or which are only called from trusted clients.

[21]Some CORBA ORBs, including OmniORB, also now support a well-known trader [OMG] – however, practical experience has shown that use of a shared filesystem is often more straightforward and reliable.

For a per-client service module, a new interface is created each time a fresh client binds to it – the collection of handles in the interface that is returned to the client refer to the closures thus generated. Thus the server may embed state that is specific to each client within these closures. Per-client services allow a server to revoke services for specific clients.

5.7 Accounting and Accountability

Since RCANE separates out the activities for each client, and uses resources accounted to each session to perform these activities, it is possible to accurately calculate the amount of each resource that a session has consumed. This allows the owner of a node to charge the end-user responsible for the session; without such charging ability, it is difficult to see how it would be practical to provide programmable servers in a network. Charging a flat rate for access to a node would be likely to result in substantial contention for the resources on the node – current network protocols such as TCP are designed to throttle back their usage of the network when they experience congestion, but enforcing such a requirement on the code supplied by users of an RCANE node would be impractical other than by explicit scheduling.

5.7.1 Pricing and Charging

Pricing and charging policies will be system dependent. They are likely to be affected by the resources available at the node, the location of the node (and hence whether it is well-connected to other computing resources) and the fluctuating demand from the remote users.

Resources charged for directly should include:

- Creating and maintaining a session.
- CPU guarantees requested.
- Actual CPU usage.
- Memory (heap, buffers, stacks, code, auxillary structures).
- Network reservations
- Network bandwidth transmitted and received.

Pricing policies should also take into account the consumption of resources that are themselves difficult to account for directly. For example, as well as consuming raw CPU cycles, a running thread will cause the processor to perform work in order to service cache misses, TLB loads, etc. Moreover, such activity is likely to cause other threads to themselves experience more cache misses, as they are forced to reload cache entries with their own data. RCANE's use of a single hardware protection domain and heavy sharing of code between sessions means that this problem is less severe than it would be when using multiple VMs, each in their own process; however, it is still likely to lead to situations in which the activity of one session may interfere with the QoS received by another session. Dealing with this problem is likely to require support from hardware, or intricate layout of heaps in order to allow limits on e.g. the amount of cache that may be used by each session. These charges may be approximated by charging more for activities (such as very frequent guaranteed access to the CPU) that are statistically likely to be associated with such crosstalk-inducing behaviour..

Possible pricing policies for resources are discussed in [Stratford99, Tschudin97b, Bredin98]. The actual billing may be accomplished in one of several ways:

- The principal could be required to hold an account with the node provider; this account would be debited directly with any usage charges.
- A credit card account (or some equivalent) could be supplied at session creation time – the node provider could make charges to this account.
- A digital payment scheme such as Millicent [Glassman95] could be employed. In such a scheme, cryptographically signed money certificates may be issued to end users through the services of a broker, and presented to the node provider as payment for resource consumption. For the reasons to be outlined in Section 5.7.2.2, digital cash that can be traced to a specific payer would be used, rather than untraceable "anonymous" digital cash.

5.7.2 Dealing with Denial of Service Attacks

In addition to performing effective resource control between multiple, possibly greedy, clients, RCANE must be capable of dealing effectively with deliberately malicious clients who attempt to commit denial of service attacks. These DoS attacks can be grouped into two main types – those against RCANE itself, and those launched from RCANE against other sites.

5.7.2.1 Attacks on RCANE

DoS attacks on RCANE could arise through malice. They could also arise due to buggy programming, for example in the event that a user accidentally supplies code that schedules an indefinite number of events, or floods the network. Thus any action that could cause consumption of limited resources should be charged for – in this way, a *denial of service* attack is effectively transformed in a *purchase of service* attack, and the attacker ends up hurting only himself. Since RCANE employs strict resource partitioning, such attacks would have no effect on any guarantees made to other clients.

In order for this to be practical, two properties are required of RCANE. Firstly, the system must be developed such that any action that a user-supplied program can take to consume resources on the node is charged for. Secondly, the system must be able to abort clients whose resources have been exhausted; this will prevent either a malicious or a buggy client from carrying on a protracted DoS attack beyond the point at which the RCANE node owner can be sure to receive payment. The architecture presented in this chapter fulfills both of the requirements, thus suggesting that RCANE should be robust against DoS attacks.

5.7.2.2 Attacks launched from RCANE

Recently, there have been many reports of "Distributed Denial of Service" attacks [CERT99], in which large numbers of poorly-secured Internet hosts have been "enslaved" to flood a particular victim – typically high-profile – site with packets, in order to overwhelm it and prevent it from serving bona-fide customers. These hosts are typically enslaved through exploiting unchecked overruns in fixed sized buffers, within server programs running with root privileges; by sending appropriate strings to such servers it may be possible to execute abitrary root-privileged malicious code on the host. A second form of attack that must be considered is one in which a malicious end-user mounts a cracking attack against a remote host, in order to make unauthorised accesses or modifications to sensitive data (such as credit card details or trade secrets) or to deface a site's web pages.

The motivation behind Rcane of permitting the execution of untrusted user-supplied code makes such attacks easier – it is no longer necessary to exploit poorly written programs in order to execute user-supplied code on a node, as the system is designed purposely to allow such behaviour!

Both of these forms of attack may be very difficult to distinguish from normal behaviour – in particular, one of the features of the distributed DoS attacks that makes detecting them difficult is the fact that the load placed on each of the enslaved machines is not necessarily abnormally high, and the effect is only seen when their streams of packets merge, much closer to the server. Thus we must conclude that detecting the launch of such attacks from the Rcane node in real-time is in general unlikely to be possible[22].

Although it is impractical to detect and prevent such attacks, there are two factors that may be used to deter potential attackers. The first is that Rcane charges for resource usage – this in itself would make including an Rcane node in a distributed DoS attack unattractive or infeasible in many cases. The second and perhaps more important factor is that Rcane needs to keep a log of activity in order to correctly charge its clients for their resource usage. This log should be sufficiently detailed to allow the node owner to trace back to an authenticated user, in the event that Rcane has been used to mount a DoS or cracking attack on another host. Without such a log, the node owner could be subject to criminal penalties for the activity carried out by the malicious end-user; the existence of such a log would help to class Rcane node owners as "common carriers" (in the same way that ISPs are not held responsible for attacks carried out over their networking services, since they can supply details of the customer site connected from a particular IP address at any given time.)

The audit log should record, for any given recent point in time, which client owned a particular local network endpoint (such as a UDP port). It is not necessary to record such information on a per-packet basis; however, it *is* necessary to ensure that a client cannot transmit a packet with header information indicating a port that the client does not own. Thus the transmission packet filters employed by Rcane are required to check any portion of the packet that is under client control.

It may be observed that such an audit log could itself be used for the type of DoS attack

[22]Note that it may be possible to use heuristics to detect certain common classes of attacks.

against Rcane described in the previous section; by repeatedly binding to and unbinding from a network connection, a malicious application could cause the audit log to grow indefinitely. In this case, since the act of writing such an entry to the audit log will only occur when a connection is opened or closed, the charge for creating the connection should take the consumption of audit log resources into account.

5.8 Summary

This chapter reviewed the fundamental principles required for the resource controlled execution of untrusted code within a programmable network, and presented the architecture used by Rcane, the Resource Controlled Active Node Environment.

The chapter opened with a discussion of the architectural principles that underly the Rcane architecture – namely that the resources used by each client in a programmable network should be accounted to the correct consumer, and that the horizontal and vertical layering of a programmable network node should reflect this requirement.

An examination of the main computing resources controlled by Rcane – CPU cycles, memory and network bandwidth – followed, along with the abstractions presented to users of an Rcane node to allow them to access and control these resources. The interaction between CPU guarantees, delayed events and incoming network packets was described.

A rationale for the use of an independent heap for each Rcane client was presented, and the requirements for the Rcane inter-session service architecture were examined, with the conclusion that the lightweight nature of protection in Rcane suggested a form of thread-migration for communication between clients.

The chapter concluded with a discussion of issues relating to the accounting and billing of remote users, and the prevention and detection of denial of service attacks.

Chapter 6

Prototype RCANE Implementation

This chapter describes a prototype implementation of RCANE, carried out in order to establish the feasibility of the architecture. It is an example implementation, and should not be considered definitive.

An overview of the implementation is given, with details of the language and operating system bases over which the implementation was developed. This is followed by discussion of some of the features of these bases that were found to be unsuitable for supporting the RCANE architecture, and how these deficiencies were remedied.

The remaining sections of the chapter describe how the major parts of the RCANE architecture were implemented.

6.1 Overview

The RCANE prototype has been developed over the Nemesis Operating System [Leslie96], described in Chapter 4. Nemesis was chosen as the base platform for RCANE for the following reasons:

- It provides good support for low-level isolation of resources and Quality of Service.
- Its low-level protection boundaries allow better control over the resources available to a program.
- The author has extensive experience with the principles and internals of Nemesis, and has been involved in development of the OS for several years.

As described in Section 5.2.1.1, RCANE consists of four layers: the Runtime, the Loader, the Core and loaded Modules.

The Runtime makes use of the Objective Caml interpreter [Leroy97] (OCaml) from INRIA to provide a safe language environment for untrusted code loaded by remote clients. Modifications were required to the OCaml interpreter to support multiple isolated heaps with independent garbage collection.

Other portions of the Runtime provide support for:

- The RCANE CPU management architecture, including real-time CPU scheduling and support for threads, thread pools and events.
- Efficient access to Nemesis I/O.
- RCANE inter-session services.
- Low-level access to session creation and heap management, for access by the Core and the Loader.

The Loader contains the RCANE bytecode linker, as well as basic session manipulation operations. It is contained entirely within the System session.

The Core provides an interface to client sessions to allow them safe access to the resources on the RCANE node. Some portions of the Core execute within the client sessions themselves. These portions are those implementing facilities for which no access to shared Core state is required. Such facilities include:

- Safe access to features exported by the Runtime for CPU and event management.
- Data-path access to network flows, both on transmit and receive.
- Safe access to Caml extensions provided by RCANE.

Since untrusted RCANE clients must supply code written in a safe language, it is not possible for clients to corrupt or otherwise abuse the portions of the Core executing within the client session.

Other portions of the Core, which do require access to shared Core state, execute within the System session. These portions include:

- Session management routines.
- Control-path functionality for managing network devices and flows.
- The bytecode librarian.

In general, data-path operations such as garbage collection, network I/O and CPU scheduling are implemented in native code in the Runtime for efficiency. Most control path operations (including bytecode loading and session creation) are implemented in Caml in the Core for flexibility, safety, and ease of interaction with clients.

The Best-Effort session uses the PLAN interpreter [Hicks98] 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, allowing straightforward control of an Rcane system. Support for demand-loaded code in the style of ANTS [Wetherall98] is also provided.

Hooks are provided to allow a security framework to process credentials and assign privileges to sessions. The focus of this implementation is on the safety, security and isolation of the untrusted code itself, rather than on cryptographic authentication/authorisation and high-level policies; therefore the default security model employed is very simple:

- Sessions and clients are either privileged or unprivileged.
- The System session and the Best-Effort session are privileged, and hence may access unsafe code sequences and routines that manage system state.
- Other sessions and external clients are unprivileged, and may only execute safe code.

6.2 OCaml Deficiencies

OCaml is an implementation of the Caml dialect of ML [Milner97]. For the implementation of Rcane, the compiler and interpreter from version 1.07 were used as a base.

The decision to use Caml was made for the following reasons:

- Caml has been previously shown to be suitable for as a language for supporting mobile code in active and programmable networks ([Alexander98a, Hicks99c]).
- Caml can be compiled to a compact bytecode that can be executed efficiently. The OCaml compiler also supports compilation directly to native code.
- The OCaml VM is small (approximately 7000 lines of code) and easily understood, simplifying its integration with the resource management mechanisms required for Rcane.
- The OCaml VM is also used as an interpreter by other source languages, such as Moscow ML [Romanenko99] – thus the choice of VM does not restrict users to a single language, provided that they can generate bytecode modules that respect the typing rules of the OCaml VM.

However, some features of Objective Caml 1.07 were found to be unsuitable for implementing Rcane. Where possible, these deficiencies have been remedied. The deficiencies fell into the following areas:

- Thread model

- Bytecode linker
- Confusion of interface and implementation
- Representation of exceptions
- Access to unsafe features
- Just-in-time compilation
- Lack of bytecode verifier

6.2.1 Thread Model

The default OCaml threading model has two implementations in the standard runtime libraries; neither implementation provides suitable support for QoS or for the event-driven processing model favoured by Rcane.

The `Thread` interface was replaced with one that supported the Rcane processing model. The new version of the interface is shown in Figure 6.1. Further discussion of the operations provided by this interface may be found in Section 6.3.

6.2.2 Bytecode linker

OCaml provides a simple dynamic linker that allows bytecode modules to be loaded into the system. However, the facilities provided are insufficient to support generalised loading of mobile code.

The major deficiency in the standard linker is that it only permits a single interface of a given name to be available for dynamic linking at any one time. If a module is loaded, any module exporting an interface of the same name (even if that interface has a different signature) is hidden by the newly loaded module. Furthermore, the namespace for locating implementations is the same as the namespace for specifying interfaces. This causes problems that will be discussed in Section 6.2.3.

Other deficiencies include:

- The lack of ability to specify the particular version of a given module against which you wish your code to be linked.
- Poor support for obtaining details of the modules linked into the system.
- No support for loading modules and then dynamically linking against symbols exported from those modules.
- No support for multiple instantiations of the same module (required to allow multiple client sessions to each instantiate a given module in their own heap).

The abstract types used by the interface.

```
type thread
type vp
type threadpool
type event
```

Create a new thread to execute the given closure in the current thread pool, the specified thread pool or the specified VP's default thread pool.

```
val create :  (α → β) → α → thread
val createp :  threadpool → (α → β) → α → thread
val createv :  vp → (α → β) → α → thread
```

Create a new idle thread in the current thread pool, the specified thread pool, or the specified VP's default thread pool.

```
val make :  unit → thread
val makep :  threadpool → thread
val makev :  vp → thread
```

Create a new thread pool in the current VP or the specified VP.

```
val pcreate :  unit → threadpool
val pcreatev :  vp → threadpool
```

Terminate the currently executing thread.

```
val exit :  unit → unit
```

Terminate the thread whose handle is given.

```
val kill :  thread → unit
```

Return the thread currently executing.

```
val self :  unit → thread
```

Return the pool of the specified thread.

```
val pool :  thread → threadpool
```

Cause this thread to go idle until more work is available for it.

```
val finished :  unit → unit
```

Suspend the execution of the calling thread for the specified number of seconds.

```
val delay :  float → unit
```

Create an event to call the given closure in the current thread pool or specified thread pool.

```
val event :  float → (α → β) → α → unit
val eventp :  threadpool → float → (α → β) → α → unit
```

Cancel the specified event.

```
val cancelEvent :  event → unit
```

Figure 6.1: The `Thread` interface

Therefore, the bytecode linker was developed into a more generalised form that permits:

- Multiple modules exporting the same interface, each of which is selectable for linking against dynamically.
- Multiple interfaces of the same name but with different signatures, which are automatically selected between at link time, using the signature imported during compilation.

Further modifications were made to the linker to support multiple clients with independent heaps, each of which may have a different set of modules instantiated in their session. This work is described in Section 6.6

6.2.3 Confusion of interface and implementation

The modifications to the linker described above permit multiple modules, each exporting the same interface, to be loaded and available in the system; however, even with this scheme it is not possible for any one module to link against multiple modules exporting the same interface, due to the confusion of interface and implementation in Caml.

Such confusion (also observed in C++ [Stroustrup86]) occurs when a language makes no distinction between an *interface type* – representing an abstract set of operations – and a *class* – representing an implementation of one or more interfaces, possibly sharing a common state. This results in the use of *abstract* classes in C++ as a representation of interface types. Caml suffers from a similar problem – although Caml *signatures* provide effective interface types, the linking model used by Caml makes it impossible to have multiple implementations of a given top-level interface type in the same program.

To alleviate this problem, run-time dynamic linking (as opposed to the load-time dynamic linking provided by the bytecode linker) was added to OCaml. A new abstract type constructor, `dynspec` (dynamic specification), was added to the compiler. This may be used to generate values that represent a particular method/variable in an specified type of interface, but with no specified implementation[1].

For example, in the context of the `Test` interface given in Figure 6.2, a dynamic specifier for `Test.number` would have type `int dynspec`, and a specifier for `Test.print` would have type `(string → unit) dynspec`. The compiler permits the textual name of an interface member to be used as a constructor for its `dynspec`.

A function, `getdyn()` (get dynamic value) was added to the interface exported by the Core to unsafe clients. By passing a `dynspec` and a reference to a loaded module, the client may obtain the member of that module's implementation specified by the `dynspec`, provided that the module does actually export the interface to which the `dynspec` refers. Thus by calling `getdyn(''Test.print'', moduleRef)` – where `moduleRef` is a reference to a previously

[1]These specifiers are internally represented as a tuple of an interface name, interface typecode and offset within the interface

An integer variable
```
val number :  int
```

A simple function
```
val print :  string → unit
```

Figure 6.2: `Test`: a simple interface for demonstration purposes

loaded module – a client may obtain a reference to the `print` function exported by that implementation of the `Test` interface, or receive an exception if the given module does not actually implement the `Test` interface. Such dynamic linking facilities are not typically available in ML.

6.2.4 Representation of exceptions

OCaml represents exceptions as tuples consisting of a *tag* (the exception identifier) followed by the exeception's parameters (if any). The tag is a reference to a string containing the name of the exception. When manipulating exceptions – which in in Caml are first class values – comparisons are made by equality of reference of the tags. For RCANE, comparison by equality is still valid within a single session. However, the multiple-heap architecture of RCANE requires a separate instantiation of each module used by a session, in that session's heap. Hence if an exception is thrown across a service invocation, such comparison of exception tags by equality is no longer valid – not only do the two different sessions each have their own version of the exception tag, but the deep copy of the exception value on an exceptional return from the service invocation would create an exception with a new tag that was equal to neither the client's nor the server's tag.

Three possibilities to remedy this were considered:

1. Compare exceptions by value rather than by reference.
2. Add a special case to the marshalling code, so that when copying an exceptional result from client to server, the exception tag could be matched with the client's version of the tag, and the correct version returned.
3. Alter the representation of exceptions so that comparison by equality was always valid.

The first solution, comparing exceptions by value, would add substantial overhead to all exception matching operations.

While the second solution would suffice for exceptions raised directly across a service invocation, the first-class status of exceptions in ML means that any data structure being passed as an argument or result may encapsulate an exception value. Thus the marshalling routines would need to be generalised to support such exception translation at any point in the copying process. Such translation would be complicated and likely to be expensive, and would need to be performed whenever an exception was raised across a service.

The third alternative – altering the representation of exceptions – was chosen since it simplified the marshalling routines. The OCaml compiler was modified to generate a different code sequence in the initialisation code for modules, so that when an exception is defined, rather than creating a string with the exception's name (which was the original behaviour) it invokes a routine in the Runtime to register the exception. The Runtime returns an atom[2] for that exception – exceptions are matched by module, exception name and position within the module. This exception registration mechanism does introduce a potential denial of service attack from a module that registers a very large number of exceptions. Such attacks are prevented by setting a limit on the number of exceptions that may be registered from each module. More elaborate protection mechanisms would be possible, but have not been implemented due to time constraints. The atom representing the exception may be passed between sessions with no further need for translation. The Runtime also defines a primitive operation for mapping an exception tag to its textual name. (Previously, the tag itself was the name.) Such mapping is only required when reporting unhandled exceptions, and thus the greater overhead due to the indirection does not affect normal system performance.

6.2.5 Access to unsafe features

Although Caml is designed to be a safe language, the standard OCaml runtime library interfaces include access to unsafe features, such as marshalling and unmarshalling of arbitrary data structures in a non-typesafe manner, and access to array elements without bounds checking.

Following the principles suggested in [Alexander98a], a module called `Safestd` was created. `Safestd` exports each of the standard library modules (giving access to abstractions such as lists, arrays and hash tables) through *thinned* interfaces – i.e. only those elements of each standard library interface that may be safely invoked by untrusted code are exported through `Safestd`. An option was added to the OCaml compiler to automatically bring the elements of the `Safestd` module into the current namespace before compiling a source file, thus permitting existing OCaml code to be compiled to use `Safestd` without requiring source changes.

6.2.6 Just-in-time compilation

To improve the execution performance of untrusted OCaml code, it would be beneficial to have a JIT (just-in-time) compiler [Cramer97] built in to the VM. A JIT compiler converts bytecode into native machine code at runtime – such conversion may be performed at the point of at which a particular function is first invoked, or it may be performed either earlier or in the background [Harris98] to avoid increasing the latency of execution.

[2]An atom is a shared system-wide integer identifier for a particular entity.

No JIT compiler for OCaml 1.07 was available. A partial implementation of an OCaml JIT compiler was developed as a proof of concept; a complete implementation was not undertaken due to time constraints.

6.2.7 Lack of bytecode verifier

In order to guarantee that a loaded Caml bytecode module actually respects the definition of the Caml language (and hence may be safely executed without need for hardware protection) it is necessary to use a verifier [Nipkow99]. The verifier checks the execution structure of the functions within the module, and ensures that the code does not perform illegal operations such as:

- Dereferencing an integer.
- Accessing the contents of an abstract data object.
- Branching to an invalid instruction.
- Accessing beyond the limits of a heap object.

This may be performed through a dataflow-like analysis of the bytecode, assigning types by inference for directly addressable locations (such as the accumulator and the stack) at each bytecode location, and ensuring that no clashes arise.

Currently there is no bytecode verifier available for OCaml. However, the bytecode used by OCaml is sufficiently similar in general style to Java Virtual Machine Language [Gosling95a] for it to be reasonable to posit that the same principles used for constructing Java verifiers could be used to create an OCaml verifier. A design outline for such a verifier is presented in Appendix A; this design has not been implemented due to time constraints.

6.3 CPU management

6.3.1 CPU Scheduling

CPU scheduling is accomplished using a modified EDF [Liu73] algorithm similar to the *Atropos* algorithm described in [Roscoe95]. Each VP's guarantee is expressed as a slice of time and a period over which the guaranteed time should be received (e.g. 300μs of CPU time in each 40ms period).

EDF is based on a dynamic priority scheme. At any point in time, the runnable VP with the closest deadline – and which has guaranteed time remaining in its current period – is given access to the processor.

The scheduler maintains priority queues of VPs in different states (Blocked with a timeout, Runnable with no guaranteed time available, Runnable with guaranteed time available).

These queues are heap-ordered to allow the scheduler to scale well with a large number of VPs.

If none of the runnable VPs have guaranteed time remaining, the processor is shared out fairly between the runnable VPs. The current scheduler walks along the queue of VPs that are eligible for best-effort time and runs the VP that has received the least amount of best-effort time since the most recent VP was created. The time required for this scan is linear in the number of VPs and hence does not scale well. However, since best-effort time is only given out when all guarantees have been met and slack time exists in the system, this inefficiency does not affect the enforcement of resource guarantees by Rcane. The choice of an optimal algorithm for scheduling best-effort time is not considered in this dissertation.

Whenever the Rcane scheduler is entered, the following sequence of events occurs:

1. Unless the system was previously idle, the elapsed time since the last reschedule is accounted to the previously running VP.
2. The next VP to be run is selected, and the period until its next preemption is calculated. From this point onwards until the time that the next reschedule occurs, all work carried out is on behalf of the new VP, and hence may be accounted to it.
3. If there are packets waiting on the owning session's incoming channels they are retrieved and transferred to the appropriate pool's packet queues. (This is described further in Section 6.5.) Any of the session's idle pools that have pending events are marked as runnable.
4. The next pool and thread to be run are selected. In the prototype implementation, scheduling between pools on the same VP and between threads in the same pool is performed using a simple round-robin scheduler. Future work would include support for client-specifed policy for pool/thread scheduling. Such support may permit selection from a number of Rcane-provided policies, and/or the ability for a client to upload its own scheduling policies in a safe way.
5. If the selected thread is currently acting as a resource backup for another thread (see Section 6.7.7), the backed-up thread is selected instead.
6. If the selected thread is active in a heap that is currently in a critical GC phase (see Section 6.4.2) then the thread carrying out the critical GC is selected instead, and will run until the critical GC activity has completed.
7. The selected thread is resumed.

6.3.2 Events and Thread pools

As described in the previous section, Rcane ensures that individual VPs receive the CPU guarantees that they request. Within a single VP, the scheduling is controlled by the client through the use of events and thread pools.

Each thread pool may be used to represent an activity being carried out by the client. By assigning thread pools to VPs, the client may control which activities have their own CPU guarantees, and which are multiplexed with other activities being carried out by that client. Assigning threads to thread pools allows the client to control the level of concurrency available to each activity. Events and incoming packet flows may be directed to particular thread pools, which correspond to the activities associated with those events and flows.

For each pool, RCANE maintains a list of events, ordered by the activiation time for that event, and a list of pending packets. If a pool has pending events or packets, and has any idle threads (or if a previously active thread completes its processing task), the next event or packet[3] is removed from its queue, and the callback function registered by the client for that event or packet flow is invoked. When the callback function returns, the thread selects a new event or packet if one is available, or else enters the idle state.

6.4 Memory management

6.4.1 Heap Management

Each client in RCANE has an independent garbage-collected heap. In general, only the threads belonging to a particular client will have access to that heap. However, when a thread makes an inter-session service invocation (see Section 6.7), it gains access to the heap belonging to the exporter of the service. To prevent the possible generation of dangling pointers (as described in Section 5.5.3), RCANE is required to ensure that two properties always hold in the system:

- No untrusted routine may have access to objects in more than one heap simultaneously.
- Every thread that has access to a particular heap must have its stack included in the root-tracing activity for that heap.

The former property is required since the garbage collectors in separate heaps are independent, and are unable to follow inter-heap pointers. Thus an untrusted routine with access to objects in multiple heaps may be able to store an inter-heap pointer in one of the objects. The latter property is required since the stacks of threads accessing a heap must be counted as roots of that heap. This could be trivially satisfied by tracing all threads' stacks when garbage collecting any heap; however, not only would this lead to unnecessary GC overhead, it would also cause substantial crosstalk between sessions (as is shown in Section 7.4).

To preserve such safety whilst maintaining performance, the prototype implementation of RCANE maintains a collection of *heap records*. Each heap record denotes a segment of a thread's stack that may contain roots into a particular heap. These heap records are organised into two orthogonal data structures:

[3]Events are given priority over packets to allow events to be delivered as close as possible to their scheduled time.

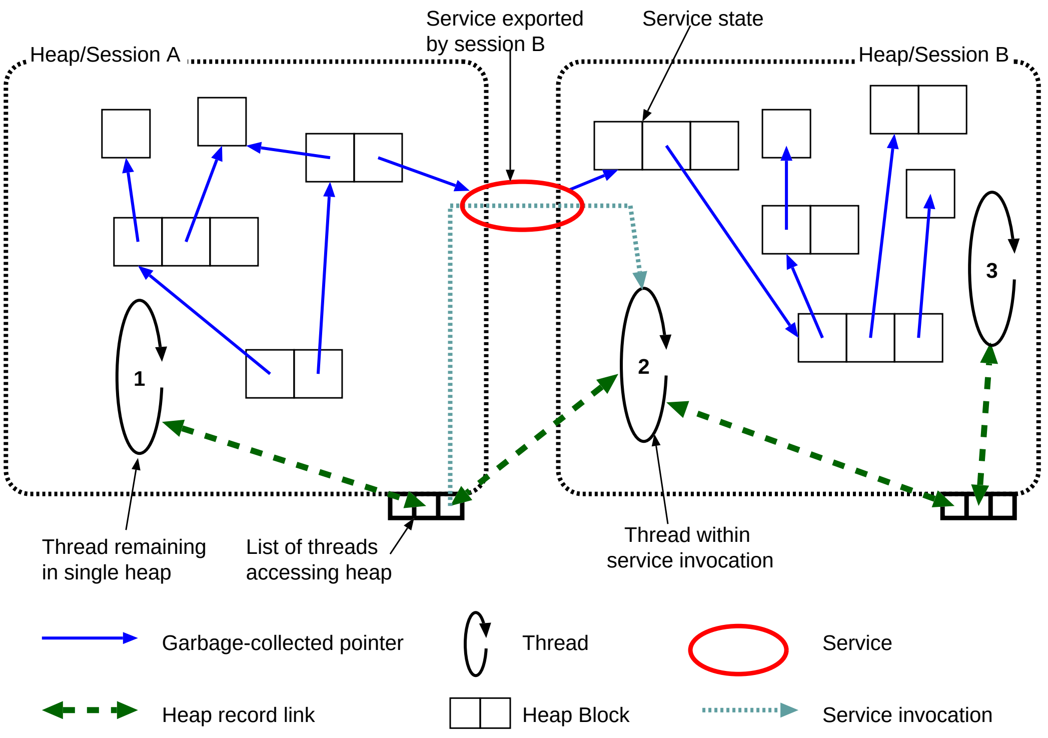

Service exported by session B
Service state
Heap/Session A
Heap/Session B
1
2
3
Thread remaining in single heap
List of threads accessing heap
Thread within service invocation
Garbage-collected pointer
Thread
Service
Heap record link
Heap Block
Service invocation

Figure 6.3: Example configuration of heaps and threads

- For each thread: a stack of heap records giving the heaps to which it has access.
- For each heap: a list of heap records giving the threads which have access to it. This list can be used to obtain the set of stack segments that must be scanned when performing root-tracing for that heap.

Figure 6.3 shows a possible configuration of threads and heaps. Thread 1 is executing purely within session A, and has stack references only to objects within A's heap. Thus its stack will only be used as a source of heap roots when garbage collecting session A's heap. Similarly, thread 3 is executing purely within session B's heap. Thread 2, however, has heap records linking it to two heaps. It belongs to session A, and hence has a heap record linked into heap A; having called through a service exported by session B, thread 2 may also have stack references to objects in heap B; thus it also has a heap record linked into heap B. When thread 2 returns from the service invocation, the heap record linking it to heap B will be removed from its stack of heap records.

Heap records are added and removed from the thread stacks and heap lists in two ways:

- Whenever a thread makes an inter-session service call, it is switched into the heap of the server. Section 6.7 describes how heap records are added and removed during service invocation.

- Routines running in the Runtime, the Loader and the Core are trusted and may switch heaps when access to a different heap is required (such as when creating a session, copying objects between heaps, or initialising a module for a session).

The interface used to control heap switching is the same for both trusted Caml code and service invocations. The `push_heap()` function pushes a heap record on to the heap stack of the current thread and links the same record on to the thread list of the heap to which access is being gained. From this point onwards, the thread will be included in garbage collection activity on the new heap, and hence may safely access objects within that heap. The `pop_heap()` function performs the reverse of `push_heap()`, popping the heap record at the top of the current thread's heap stack and removing it from the heap's list of accessing threads.

A reference to the currently active heap is made available to the trusted portion of the virtual machine – this reference is an l-value, and hence trusted code executing in the virtual machine may switch heaps without the expense of making a function call into the Runtime. (Prior to this, `push_heap()` should have been called on the heap being switched to.)

6.4.2 Garbage Collection

The garbage collector is based heavily on the OCaml collector – the fundamental garbage collection algorithm used is unchanged from that used by OCaml 1.07.

Each session has its own independent two-generation heap. The younger generation (*minor heap*) is collected via Stop/Copy into the older generation. The older generation (*major heap*) is collected with an incremental Mark/Sweep algorithm. Most allocations are made from the minor heap, but native code in the Runtime may allocate large objects – or objects that are expected to be long-lived – directly in the major heap.

[Doligez93] describes the GC algorithm used by *Concurrent Caml Light* (CCL), a different dialect of Caml that shares many characteristics with Objective Caml. The two-generation scheme and shading invariants used in the CCL collector are the same as in Objective Caml. The CCL collector differs in that it provides each thread with its own minor heap and requires regular synchronisation between the garbage collector thread and the mutators – since each Rcane client has only a single heap, these concurrent portions of the CCL GC algorithm are not relevant to Rcane.

If we wish to avoid using a write barrier on stack locations when tracing the roots of a session's heap, it is necessary to suspend all threads that might access that heap. To ensure that all appropriate threads are suspended during such *critical* GC activity, the heap records described in Section 6.4.1 are used to allow such suspension to be performed easily and efficiently.

The top heap pointer on each stack is the thread's *active* heap, indicated through the use of a field in the heap record. (For brief periods of time, such as 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 that are active in the heap are suspended, other than to carry out the GC work. Such suspension is implicit – since there may be many threads active in a heap, actually suspending and resuming all threads in order to perform a small amount of root tracing would be excessive. Instead, if a thread's active heap is in a critical GC phase when a thread is scheduled to run, execution is instead diverted to the thread performing the critical GC activity, in order to complete the critical phase as quickly as possible.

The majority of the Mark/Sweep activity can be carried out without halting any of the threads. Currently a portion of Mark/Sweep activity is carried out after each collection of the young generation. The amount of work done in each Mark/Sweep phase may be tuned by the session owning the heap.

Tracking the threads that have access to each heap minimises the number of threads' stacks that must be traversed to identify roots, and prevents QoS crosstalk between principals that 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 (untrusted) code running in the server cannot access that heap. Furthermore, any pointers to objects in the client's heap that are stored in enclosing stack frames will not be accessed until the thread returns from the service, since Caml does not permit dynamic scoping of identifiers[4]. Thus a thread need not be suspended in order to perform root tracing on heaps to which it has access but in which it is not marked as active. If a thread becomes active in a heap in which critical GC activity is being performed (such as when invoking or returning from a service) it is suspended until such activity has completed.

The majority of GC activity is not classed as *critical* – one thread may be performing Mark/Sweep activity on a heap concurrently with other threads accessing objects within that heap. A write barrier is used on heap objects to ensure correct concurrent operation.

6.4.3 Low-memory conditions

Gracefully handling low-memory and out-of-memory situations is a difficult challenge, and one that is often met poorly by application and systems writers.

Garbage collection can make recovery from such events easier, since it is possible for exceptions to be raised and left unhandled without leaving large swathes of unreclaimable memory that exacerbate the shortage of memory.

However, the garbage collection can itself obscure the details of memory management from the programmer, particularly in languages such as ML in which memory allocation is often implicit. Typical garbage collecting allocators will increase the size of the heap (up to some large system-wide maximum) if insufficient free space is available to satisfy an allocation request, since it is generally the case that the application is working on behalf of the owner of the machine, and hence should have access to as much memory as it requires. When running

[4]If a language such as LISP [Steele84] or Perl [Wall87] – which do allow access to identifiers to be made through the dynamic call chain to higher stack frames – were to be used as the safe language for an Rcane system, such a guarantee would no longer be possible.

on RCANE, in which relatively small per-session limits on memory consumption may be made in order to prevent excessive resource consumption by remote principals, the limit on the size of the garbage collected heap may be reached somewhat sooner. Possible courses of action in the event of a low-memory condition include:

- Abort the session.
- Return a null pointer.
- Raise an exception.
- Call a programmer-provided "low-memory" hook.

Aborting the session is clearly undesirable, and should be kept as a last resort. Returning a null pointer in Caml is not possible, since a fundamental feature of the ML family of languages is that all pointers are always valid – a constructor cannot return a null pointer. This allows substantial optimisations without compromising safety. Raising an exception is a valid operation at most times, and indeed raising `Out_of_memory` is supported by the OCaml garbage collector for the rare occasions in which it cannot obtain memory from the underlying OS; however, this has certain drawbacks:

- Since any non-constant[5] constructor may thus raise an exception, either the programmer must explicitly check for the `Out_of_memory` exception at a great many points in the code; or else an `Out_of_memory` exception must be treated as a generic abort signal potentially causing the session to terminate under programmer control.
- In a multi-generational heap, memory may become exhausted whilst copying from a newer to an older generation. If this occurs, the state of the heap may be such that it is not safe to continue execution of the session, thus requiring termination of the session.

Low-memory situations in RCANE are handled with a combination of a callback, a buffer zone and the `Out_of_memory` exception, as described in the following paragraphs.

To avoid the situation where heap memory is exhausted during a copying phase from the minor heap to the major heap, RCANE maintains a free buffer zone within the major heap that is at least as large as the minor heap; thus even if the entire minor heap is copied into the major heap during a minor collection – an event that generally does not occur – heap memory will not be exhausted.

If, following a copying phase from the minor heap to the major heap, this buffer zone has been encroached upon, a user-registered callback function is invoked to release memory. This function may employ any desired strategy to reduce memory usage, so as to allow the requested allocation to complete. For the duration of the callback function, other threads attempting

[5]ML constructors may be divided into *constant* and *non-constant* constructors. Constant constructors represent integers, literals and parameterless members of algebraic types. Non-constant constructors represent tuples, records and parameterised members of algebraic types, which require memory allocation.

to allocate memory from that heap are suspended. Following the completion of the callback, a garbage collection pass is made. Memory allocations made directly from the major heap (which occur only for large objects, and never due to implicit constructors) that would eat into the buffer zone cause an `Out_of_memory` exception to be raised; this may be dealt with by the user.

This strategy is clearly not perfect – in particular it does not address the following concerns adequately:

- There is no way for the callback function to know the importance (to the client) of the requested memory allocation. This could perhaps be achieved by passing the Caml program counter at which the allocation was made, and allowing the callback to map this in some way to a function identifier or source code line. Such a facility would need support from the compiler and the OCaml runtime, along with substantial investment and maintenance from the programmer.
- The action that may be taken within the recovery callback function is limited, since if excessive memory is allocated during the callback, the session is by necessity aborted.

However, the approach taken by Rcane does have the benefit that it is the clients that must consider the implications of memory exhaustion, rather than the Rcane node itself. Since the clients are in control of their memory allocation strategy, it is more straightforward for them to perform such recovery considerations.

6.5 Network I/O

Rcane provides access to the network through the abstraction of channels. A channel is a connection to a device driver associated with a particular network card[6]

Rcane channels are implemented over Nemesis I/O channels. For clients with no network resources reserved, multiple Rcane channels may map to the same Nemesis I/O channel; if the client has reserved resources, the Rcane channel will have a dedicated I/O channel.

Each channel may be used to send and receive packets on a set of flows, specified by packet filters. These packet filters are installed by Rcane in the Nemesis device driver when the channel is created.

The prototype currently supports two classes of channels: session-based virtual networks and channels for access to local network endpoints.

Session-based virtual networks: A virtual network may be set up – as part of the configuration of an Rcane node – either for a particular Ethernet frame type or for a

[6]In the case of a UDP channel associated with the `INADDR_ANY` (wildcard) IP address on a node with multiple network cards, the channel may transparently encapsulate multiple such connections.

Field	Meaning
protocol header	Header for encapsulating protocol (UDP or Ethernet)
destination	Ultimate destination of packet
resource bound	Resources associated with packet
session ID	Network-wide identifier for sending session
flow ID	Demux identifier for sub-flow

Figure 6.4: The header format for RCANE virtual network packets

particular UDP port (with a given set of neighbours). Each packet sent on a virtual network channel has a packet header as shown in Figure 6.4. This header format is a prefix of the PLAN [Hicks99c] packet header format, allowing straightforward use of PLAN interpreters and interaction with existing PLAN nodes. The *session identifier* field (see Section 6.8.1) within the packet header is used to demultiplex the packet to the appropriate session; the *flow* field may be used to further demultiplex the packet if the session has specified handlers for multiple sub-flows of packets within the channel.

Local network endpoints: Sessions may open channels to receive packets on network endpoints on the local node, in a similar manner to traditional *sockets* APIs. Policy for access permissions to packet flows – such as restricting unprivileged sessions to bind only to ports greater than 1024 – may be specified on a per-node basis. Packets received on local network endpoints need have no particular packet format beyond that required for the specification and demultiplexing of the packet (e.g. a UDP header).

Link-level frames/packets are classified on reception by the network device drivers via the network packet filter, which maps the packet to the appropriate channel. Packets for local network endpoints are routed directly to the channel specified for that endpoint. Packets received on virtual network connections are classified as follows:

1. Packets for sessions that have reserved guaranteed resources on the given virtual network device are routed to the channel created for that session.

2. Packets for sessions that have registered on the receiving Rcane node, but which have not reserved guaranteed resources on the given virtual network device, are routed to the channel created for the Best-Effort session.

3. Packets for sessions that have not registered on the Rcane node are routed to the channel created for the Best-Effort session.

If the channel has free buffers available, the packet 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 VP associated with a given channel is scheduled by RCANE to receive CPU time, the packets are extracted from the channels and demultiplexed to the appropriate thread pools. At this point, the following operations are performed:

1. First, any required processing for the encapsulating protocol (such as UDP or Ethernet) is performed. After this stage, protocol processing for a packet received on a local network endpoint is complete, and the packet is added to the event queue for the thread pool specified by the receiving session. Further processing is required for packets from virtual network devices, as described in the following stages.
2. If the packet is destined for a session registered on the receiving node, but has been queued on the channel belonging to the Best Effort session (due to the receiving session not having reserved the appropriate network resources) it is at this point demultiplexed to the destination session.
3. The flow identifier in the packet header is used to demultiplex the packet to the appropriate thread pool and handler function, as specified by the receiving session.

Such demultiplexing is performed within the Runtime for efficiency, and is accounted to the appropriate VP (since it is only performed when that VP has processing time available). Once the thread pool and handler function that are to be used to handle the packet have been selected, a callback event is constructed that invokes the handler function. The arguments to the handler function consist of the packet contents and the channel on which the packet was received.

Once the client's callback has finished processing the packet, the buffer memory is returned to the underlying Nemesis I/O channel, to be reused for future packets.

Transmit scheduling is performed by the Nemesis device drivers following a modified EDF algorithm, as described in [Black97]. Each client channel is given a transmission scheduling period and slice of transmission time, in a manner analagous to the scheduling of CPU time. The network device driver repeatedly selects the next "runnable" channel that is eligible for network time, sends a packet from that channel, and charges the channel for the time that was taken up by the packet on the network link[7], rather than the CPU time consumed by the driver in sending the packet (which is negligible when using a network card with effective DMA capabilities). Thus on a 100Mb/s Ethernet, a 1KB packet would be charged for 80μs of link time. A channel is considered "runnable" when there are packets queued awaiting transmission.

6.5.1 Enhancements to Nemesis network subsystems

In general, Nemesis was found to provide a good match for the networking interface required by the lower levels of RCANE's Runtime. The low-level protection boundaries of Nemesis allow RCANE to enhance the standard Nemesis networking abstractions to ensure that network processing is only performed for a given client when that client's VP has CPU time available.

[7]Note that the current implementation does not take into account link time wasted due to packet collisions.

Extensions were required in the Nemesis generic network driver top-half to provide support for session-based virtual networks and demultiplexing of raw Ethernet frames.

6.6 Bytecode Librarian

The bytecode librarian runs as part of the Loader, and is responsible for linking in modules of bytecode, and keeping track of

- which modules a given session is privileged to link against,
- which sessions have linked against each module,
- the dependencies between modules, and
- the total amount of code memory consumed by each session.

OCaml stores the global data of each module in a heap record, and stores a pointer to that record in the module's entry in the *global array.* Since each session in RCANE requires its own copy of a module's global data items (in its own heap), RCANE provides each session with its own global array. If a session has linked against a particular module, then that session will have a valid pointer to the given module's global data in the appropriate slot of its global array, otherwise the slot will be null. This record is created at module initialisation time – thus it is important for RCANE to ensure that a session not be allowed to execute code from a module before that module's dependencies have been initialised.

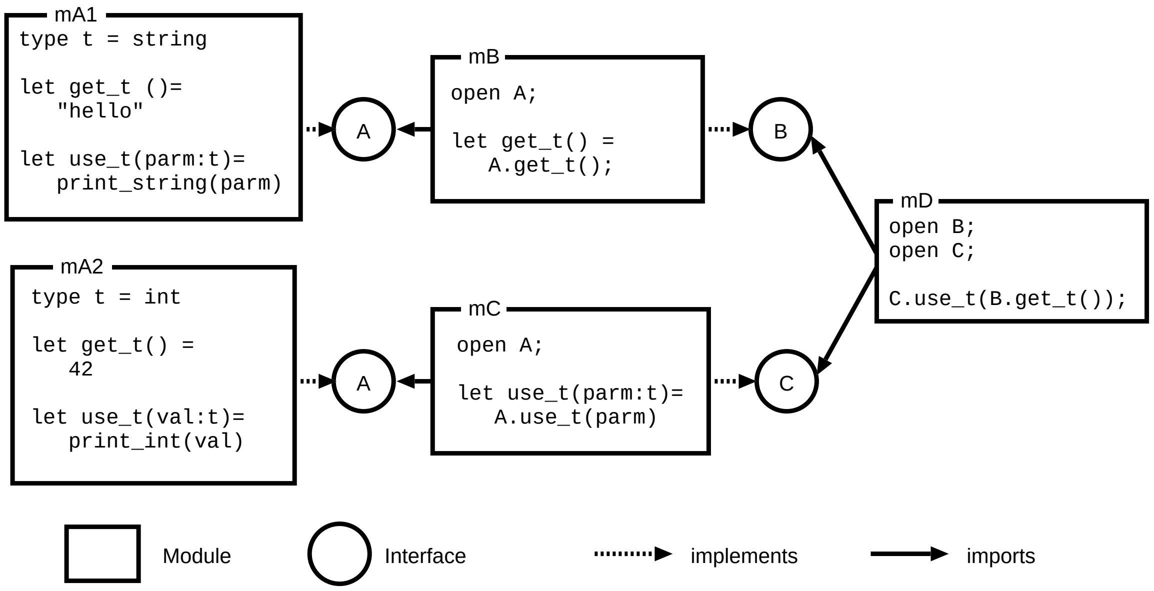
When a session links against a particular module, the code librarian first performs a depth-first traversal of the module's dependencies, and links any previously unlinked dependencies. Next, the initialisation code for that module is executed (in the context of the client's heap), to create the *global entry* for the module. The global entry consists of an array of closures and values corresponding to the elements exported by the module's interface, and is stored in the appropriate slot in the client's global array.

Note that the procedure for initialising Service Modules – used to export an entire interface of services from a server to a client – differs from the above description, and is more fully discussed in section 6.7.5.

6.6.1 Linking

When a new module is to be linked into the system, the following steps are performed:

Unmarshalling: The module is initially presented as a binary object. This object must be unmarshalled to produce a structure describing the module.



```
type t = string

let get_t ()=
   "hello"

let use_t(parm:t)=
   print_string(parm)
```

```
open A;

let get_t() =
   A.get_t();
```

```
open B;
open C;

C.use_t(B.get_t());
```

```
type t = int

let get_t() =
   42

let use_t(val:t)=
   print_int(val)
```

```
open A;

let use_t(parm:t)=
   A.use_t(parm)
```

Figure 6.5: Potential type safety violation

Dependency Checking: Each module will have a set of statically imported interfaces. For each such interface, there must have already been linked a module exporting that interface. A particular import module may be specified for a given interface[8] to deal with the situation where multiple modules export the same interface. If any of the imports are unavailable, the link stage fails.

Furthermore, since opaque type definitions may be concealed behind Caml interfaces, it is necessary to put the following constraint on the linkage tree of modules:

No module is permitted to refer to more than one module exporting any given interface.

The relation `mA` *refers to* `mB` is true when:

- Module `mB` exports interface `B`,
- `mB` is in the transitive closure of module `mA`'s imports, and
- there exists an import path between `mA` and `mB` in which all modules make use of types declared in interface `B`.

Such a constraint is required due to the following possible scenario, shown graphically in Figure 6.5:

1. Interface `A` declares an abstract type `t`, with operations `get_t()` (to return a value of type `A.t`) and `use_t(t)` (to process a value of type `A.t`).
2. Module `mA1` implements interface `A`, privately defining type `t` to be `integer`.
3. Module `mA2` implements interface `A`, privately defining type `t` to be `string`.

[8]A given implementation may be specified by presenting a hash of its code.

4. Module `mB` links against module `mA1` and implements interface `B`, which defines an operation `get_t()`.
5. Module `mC` links against module `mA2` and implements interface `C`, which defines an operation `use_t(t)`.
6. Module `mD` links against modules `mB` and `mC`. It may thus call `C.use_t(B.get_t())`, passing a value of type `integer` to a routine expecting a value of type `string`.

This constraint is related to the requirement to prevent *bridge classes* [Saraswat97] in Java from creating similar type system violations.

Verification: The bytecode in the module must be shown to conform to the interfaces which it imports/exports, and to respect any typing constraints associated with the language in which it is written, using techniques such as those presented in [Nipkow99]. Appendix A presents a design outline for a verifier for Objective Caml bytecode.

Resolution: The imports of a module must be fixed up so that they point to the corresponding exports in the modules that are being imported.

6.7 Service Implementation

This section presents some of the key features of the implementation of the Rcane inter-session service architecture.

6.7.1 Overview

When adding services to Rcane, two main possibilities were considered:

- Extending the Objective Caml bytecode language and virtual machine definition to provide specific instructions for invoking services.
- Adding support for services through the use of additional primitive functions written in native code.

Since the Caml typing rules do not make it practical to identify service invocations at compile time, the latter option was chosen.

6.7.2 Representation

There are five components to an Rcane service:

- The *service function* is the actual function that the server session wishes to export to the client, across an existing service binding.

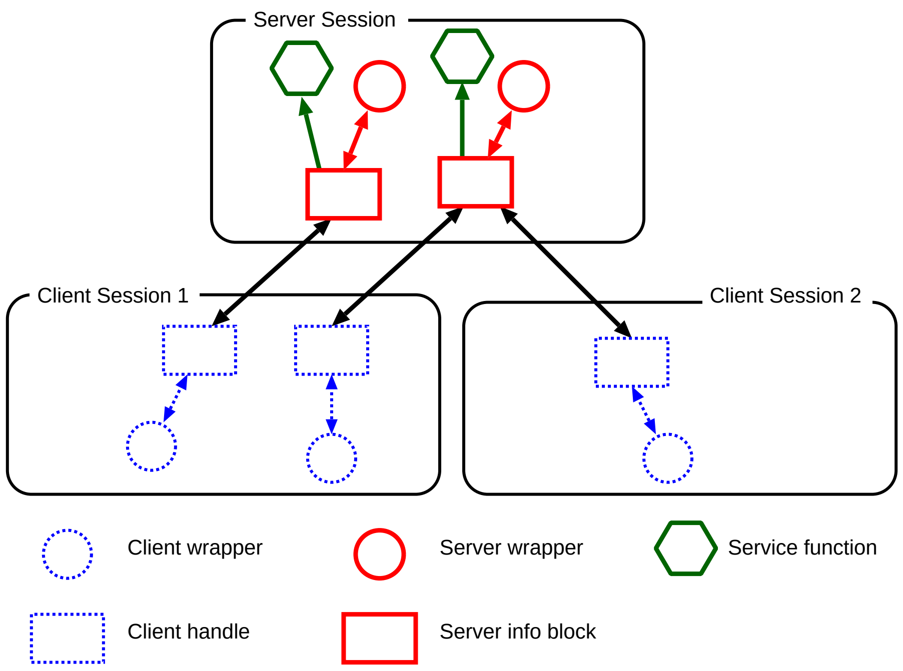


Figure 6.6: The representation of services in RCANE

- The *server info block* contains all the server-side state required by RCANE to implement the service. Within a service's info block, a table is maintained with entries for each client that holds a handle for that service.

- The *server wrapper* is a function closure that is a copy of the service function; thus it could be called from within the server to obtain the same effects as calling the original service function. However, it has two additional properties:

  – It is tagged[9] so that the inter-session copying mechanisms can identify it as a service and copy it correctly (as described in Section 6.7.4).

  – It contains additional information in the *environment* portion of the closure to allow its info block to be located.

  The server wrapper is the component that the server passes to the client.

- The *client handle* contains all the client-side state required by RCANE to implement the service.

- The *client wrapper* is a closure of the same type as the server wrapper. However, invoking the client wrapper causes the service mechanism to be entered, and control transferred to the service function in the server's heap (see Section 5.6.4).

[9]Every object in the OCaml heap has a *tag* as part of its header providing information about the object type.

Figure 6.6 shows the situation where a server session exports two services. Client 1 has bound to both services; client 2 has bound only to one service.

6.7.3 Type-based Optimisations

In the OCaml heap, each object is either *abstract* (not traced internally by the garbage collector, e.g. strings) or else contains a series of *values.* Each value is either an integer or a pointer, distinguishable through checking its LSB[10]. This allows the garbage collector and the RCANE parameter copier to reliably trace objects through memory without risking confusing pointers and non-pointers.

By default, parameters to RCANE services are copied by tracing the parameters' structures in memory, and making copies in the new heap. In the case that the server has specified the `Graph` attribute for the service, indicating that a parameter being passed may contain shared structures, the copying routine additionally maintains a list of previously encountered objects, and ensures that pointer sharing in the source heap is preserved in the destination heap.

However, as will be demonstrated in Section 7.3, this generic copying mechanism incurs overhead due to the checking and bookkeeping that it must perform.

The nature of ML's algebraic type system tends to conspire against fully optimising the copying routines. An algebraic type has multiple constructors, some of which may be constants (represented as integers) and others of which may be some kind of object (represented as pointers). An example of such a type, representing a tree containing strings and integers, might be:

```
type tree = StringNode of string * tree * tree
          | IntPairNode of int * int * tree * tree
          | Leaf
```

Thus a value of type `tree` might contain:

- a pointer to a three element block with the tag `StringNode` containing a string pointer and two `tree` values,
- a pointer to a four element block with the tag `IntPairNode`, containing two integer values and two `tree` values, or
- the integer value `Leaf`.

[10]This has the result that OCaml integers can hold one less bit than a machine word.

Hence, it is impossible to fully determine until when actually performing the copy, what the structure of a `tree` parameter will be. However, in many situations it is possible to make guarantees about the type of a parameter and thus substitute an optimised copying routine.

In a similar manner to the dynamic stub generator developed by the author for Nemesis IPC (see Section 4.4 and [Menage98a]), RCANE analyses type information about dynamically loaded modules, to identify situations in which the types of parameters are known in advance.

A full implementation of RCANE would employ a mechanism similar to the Nemesis stub generator: firstly the type for a parameter would be converted into a simplified representation, and secondly this representation would be used to synthesise (or reuse) a suitable machine code function.

Such type analysis is performed in the prototype, but time constraints have prevented implementation of fully synthesised copying routines. As a proof-of-concept, any parameters that can be shown to be:

- blocks containing only integer values, or
- integer lists

are copied using optimised copying routines. Note that since ML represents all non-pointer types as integers, any algebraic type with purely constant constructors may be regarded as equivalent to an integer for marshalling purposes, increasing the range of possible optimisations.

6.7.4 Copying Services Between Sessions

Passing services (either server or client wrappers) between sessions as the parameters or results of a service invocation is used as a mechanism to allow clients to bind to services. The tag on a memory block for a server wrapper or client wrapper identifies it as such to the parameter copier. Rather than directly copying the contents of the wrapper to the destination heap, instead it acts as follows:

1. The info block for the service is located via the pointers available in the client or server wrapper.
2. If the destination heap belongs to the server, a pointer to the existing server wrapper is returned as the result of the copy operation.
3. If the destination heap belongs to an existing client of the service, a pointer to the existing client wrapper for that client's session is returned as the result of the copy operation.
4. Otherwise, a handle and client wrapper for the service are created in the destination heap, and added to the info block's client table.

In this way, services may be passed around by reference as parameters and results of service invocations.

6.7.5 Service Modules

The service passing described above allows services to be passed across pre-established service bindings. To "bootstrap" initial bindings between a client and a server, and to provide a convenient way of binding to a standard set of services supplied by a server, RCANE supports *service modules*, as described in Section 5.6.6. Section 6.6 described how modules are linked and initialised in a session. This section describes how the procedure differs for service modules.

Each service module is associated with a particular server. When a session binds to a service module, rather than executing the initialisation code of the module in the context of the client's heap, the initialisation occurs in the server's heap. Every element in the global entry of a service module must be a service. Rather than simply storing the global entry in the global vector of the client session (which would result in an inter-heap reference), the entry is copied using the same mechanism as is used for service invocation parameters. This has the effect of creating handles for all the exported services in the client's session. Figure 6.7 shows how the services exported by a service module are made available to the client session. The global entry for the service module (visible to RCANE clients as the interface exported by the service module) hangs off the client's global vector. In the server's session, the various services typically share some or all of their state.

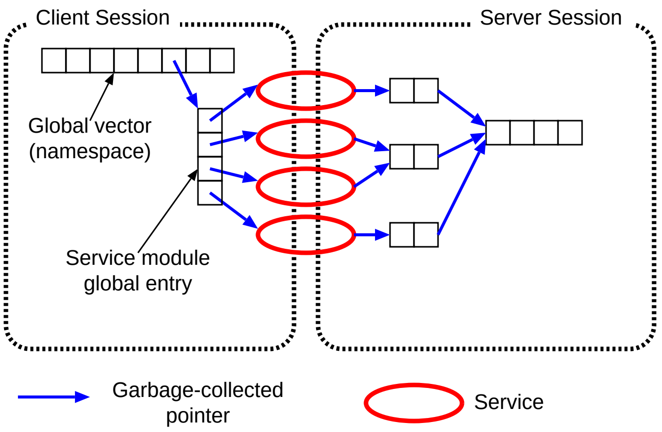

Client Session
Server Session
Global vector (namespace)
Service module global entry
Garbage-collected pointer
Service

Figure 6.7: Example of a service module

A service module may be classified as having per-client state or shared state. The initialisation code of a per-client service module is executed when each new client binds to it – thus a new set of services are created that are associated specifically with the client. A shared service module is initialised once, when the server loads it. This creates a set of services that are copied to each client that binds to the service module, with the result that all clients share the same set of services. A server that wishes to access both shared and per-client state for a service may create two modules; the first module is a per-client service module that actually creates the desired services, and the second is an ordinary module bound in the server's session, containing the desired shared state. The service module imports the state module, giving the per-client services access to the shared state.

6.7.6 Garbage Collection of Services

At creation time, a service is specified as being either `Retained` or `Collected` – this allows the server to control whether the service is kept alive as long as any clients hold a handle on the service, or only while the server itself maintains a reference to the info block. For any service that is specified as retained, RCANE maintains a heap root for that service's info block while clients are bound to the service, to ensure that the service does not get garbage collected whilst it is in use.

When a service info block is eventually garbage collected, all client handles for that service are marked as being revoked, causing future invocations using such handles to raise a `Revoked` exception.

If the client handle becomes unreachable and is garbage collected[11], the handle's entry in the service's client table is removed.

Either the server or the client may explicitly force a service or a handle to be released (by calling `Service.destroy()` or `Service.release()` respectively) without requiring the service or handle to be garbage collected first. Further attempts to use a handle or service that has been explicitly released cause an exception to be raised.

6.7.7 Abnormal Service Termination

Not all service invocations can be assumed to complete correctly. Since the client or server sessions may exit at any time, or be terminated due to exhaustion of resources, RCANE must be able to support abnormal termination of service invocations.

A structure describing each service invocation is linked in to a per-thread list and a per-service list for the duration of the invocation. These lists allow RCANE to identify:

- the threads belonging to a session that are currently performing service invocations to other sessions (in the event of a client exiting, or aborting an invocation), and
- the threads currently executing in a session's heap that are in service invocations from other sessions (in the event of a server exiting, or revoking a service).

If a service invocation is required to be terminated, the affected threads may be located through these links, and exceptions raised at the point of the service invocation.

[11]This will not happen whilst the client maintains a reference to the client wrapper, since the client wrapper contains a reference to the handle.

Additionally, to allow servers to deal with the case where a client terminates whilst executing in a critical region in a service invocation, it is possible to register *backup* threads for critical regions that are executed as part of a service invocation (discussed in Section 5.6.2).

A backup thread allows a server to guarantee a certain level of resources – specifically, CPU time and stack space – to a client thread, to prevent a CPU-starved thread from causing QoS interference through spending too long in a critical section, or to allow the execution context of a terminated thread to continue until it has exited the critical region.

In the prototype implementation, backup threads are assigned to client threads by allowing a thread pool to be associated with a mutex; when the critical region protected by the mutex is entered, and the mutex is locked, an idle thread from the associated thread pool is assigned to be a backup thread for the client thread. If no idle thread is available from that pool, the locking thread blocks until a thread is available. When the client exits from the critical region, the backup thread is made idle again.

The existence of a backup thread has two significant effects on a client thread:

- The backup thread is scheduled like any other thread. However, when a backup thread is due to be run, rather than actually running the backup thread (which has no execution context, having been idle before being assigned to the client thread) the scheduler instead runs the client thread. Thus the client thread receives CPU time from two sources – from its own VP's CPU guarantee, and from the CPU guarantee provided by the server to the backup thread. Therefore client threads with sufficient CPU time are not limited by the level of CPU resources assigned by the server, whilst the server can bound the level of crosstalk caused by the critical region through assigning an appropriate level of resources to the backup thread pool.

- In the event that the client thread is destroyed or the client aborts the service invocation, the execution context (stack, registers and auxillary state) of the thread is transferred to the backup thread. From this point on the backup thread acts as a normal thread, continuing the execution of the critical section. When the critical section is completed, the backup thread returns to the idle state.

6.7.8 System Services

Rcane uses a basic set of services to allow clients to communicate their requirements to the System session.

The main interface is the `Session` interface, implemented as a per-client service module. Each instantiation of this module maintains the state about a single session. Additionally, the `Session_priv` module in the System session provides auxillary routines and shared state.

As described in Section 5.2.1.2, the `Session` interface provides the primary point of access for requesting QoS guarantees (for CPU, memory and network access), loading/linking code modules, and session creation/termination.

To support access to the network, the `Rip` and `Arp` service modules provide routing and address resolution services. These modules were ported largely unchanged from PLANet [Hicks99c] – some adaptation was required to fit into the resource-isolation model supported by RCANE.

6.8 Session Creation and Destruction

New sessions are created by invoking the `Session.createSession()` service exported by the System session. Creating a session involves the allocation of a heap and a default VP (with a single thread and best-effort access to the CPU) as well as Runtime housekeeping data for the new session. RCANE aggressively caches objects such as heaps, threads and VPs to allow session creation and destruction to be very lightweight. When a session is created, credentials are passed to authenticate the entity that will be responsible for the session, and to provide a mechanism by which RCANE may bill that entity for resource consumption (see Section 5.7).

Since RCANE aims to charge each session for the resources that it consumes, as much of the session creation work as possible should be undertaken on a VP owned by the newly created session, rather than on one owned by the creating session or the System session. When a session is created, it initially has no modules linked and no data. To bootstrap the newly created session, the `SessionBoot` service module, which is exported by the System Session, is bound into it – this copies a single service, accessible as `SessionBoot.bootstrap()`, into the new session. A callback event that invokes `SessionBoot.bootstrap()` is registered for the session's VP; at this point all further session creation work is performed by the session itself.

Upon invocation of `SessionBoot.bootstrap()`, the code librarian loads/links the initial module specified (in the call to `Session.createSession()`) for the new session. This will in turn link in any dependencies of the initial module – typically these will include `Safestd` (the standard library) and the `Session` interface itself.

When destroying a session (either through resource exhaustion or at its own request), the following sequence of events occurs:

1. All client handles held by other sessions for services exported by the dying session are marked as revoked. All client threads from other sessions that are in service invocations to the dying session are jumped back into their calling sessions; in each case a `Revoked` exception is raised at the point of invocation.

2. All of the dying session's client handles are marked as revoked. All threads belonging to the dying session that are in service invocations to other sessions are jumped back into the dying session.

   At this point, the dying session is completely isolated from other sessions – it has no handles to other services (even to the System session), and no other sessions have handles on its exported services.

3. The session's networking resources are revoked.

4. The session's threads are all terminated.
5. Housekeeping information maintained by the System session is released.
6. The Runtime structures and heap are released.

6.8.1 Session Identifiers

Associated with each session is a network-wide unique *session ID*, composed of the network address of the node at which the session originated, and a unique identifier within that node.

When a new session is created at a node, it may either be associated with the session ID of the remote client that created it (provided that the remote client has not already created a session at the current node), or may have a fresh session ID generated, with the current node as the originating node stored in the session ID.

It is intended that this session ID eventually be used to support resource transfer across multiple programmable nodes, thereby potentially reducing the billing overheads associated with programmable networks. Such support is currently limited – the *resource bound* field in RCANE virtual network packets may be used to transfer resource units between sessions owned by the same client on different nodes. These resource units are not yet fully integrated with the RCANE resource accounting mechanisms.

6.9 Summary

This chapter has described the key implementation details of a prototype of the RCANE architecture built over the Nemesis Operating System.

For the prototype implementation, Objective Caml was chosen as the language and Nemesis as the platform for RCANE. The rationale behind these choices was discussed, and a number of deficiencies in OCaml were identified; in each case the deficiency was remedied, or else parallels with existing research were presented that suggested that a remedy for such a deficiency was practical.

Although the prototype implementation was developed over Nemesis, the RCANE architecture could be implemented over any OS providing suitable support for QoS isolation; similarly, any language with suitable safety properties could be used as the client language.

The chapter then discussed implementation details of the major features of RCANE. In each case the design decisions that promoted effective QoS isolation and performance were presented. The EDF scheduling algorithm provides the CPU guarantees required by the RCANE architectures; the abstractions of events and thread pools, which provide an interface for the CPU scheduler to clients, were discussed. To provide the separated heaps that allow RCANE to prevent crosstalk due to garbage collection, RCANE must keep track of the sets of threads with access to each heap, and must ensure that inter-heap references cannot be created by untrusted clients. The implementation employs a similar mechanism for network transmission as for CPU scheduling; this allows effective transmission guarantees to be made. Packets received over the network are demultiplexed as early as possible in order to prevent crosstalk between different clients' packets.

Inter-session services support lightweight communication between clients executing on Rcane; several implementation details of services were described: their structure, the optimisations employed by Rcane to improve copying performance and the mechanisms used to ensure safety in the event of abnormal termination.

Also discussed were two control-path features of Rcane: the bytecode librarian and the mechanisms for registering and deleting clients' sessions.

Chapter 7

System Evaluation

This chapter and Chapter 8 present the results of experiments carried out to test the prototype implementation of RCANE. Aspects of the architecture are considered in terms of performance, resource isolation, and flexibility.

The evaluation is structured into two parts. In this chapter, the basic RCANE system is assessed – the results of micro-benchmarks are presented for the key features of the architecture presented in Chapters 5 & 6. Chapter 8 examines the use of RCANE as a base architecture for active network systems.

7.1 Evaluation context

7.1.1 Experimental equipment

The majority of the experiments presented in these chapters were performed on Intel Pentium II machines running at 300 MHz connected to 100Mb/s Ethernet. Some experiments also involved Intel PentiumPro machines running at 200MHz, connected to 10Mb/s Ethernet.

7.1.2 Instrumentation

The RCANE prototype was instrumented in order that events of interest (for use during debugging and performance measurement) could be recorded in an in-memory log and later flushed to disk for analysis. Events recorded in the log included:

- Scheduler entry/exit and scheduling logic
- State transitions for threads, thread pools and VPs
- Garbage collection activity

- Packet transmission and reception

Logging of specific types of events was made configurable at compile time, allowing logs to be kept only of the events of interest to a particular experiment, and preventing any wastage of CPU cycles due to unwanted events being logged.

Events were recorded with cycle-time accuracy; the time taken for a single event to be logged was found to be approximately 0.2μs.

7.2 Scheduler Performance

The fundamental advantage of a programmable network over a passive network is that end-users may perform computations at nodes within the network, and hence receive lower latency than if all interactions were required to take place between the end-user and the ultimate destination. In order to provide low latency, it is vital that the user-supplied applications running on the programmable node receive timely access to the CPU. Alternatively, if the application is attempting to process or filter some form of multimedia data, it should receive regular access to the CPU in order to prevent excessive jitter in its results. Thus an important feature of RCANE that must be shown is that it allows users to request and receive the guaranteed access to the CPU that they require.

To demonstrate the effectiveness of the CPU guarantees provided by RCANE, the time received by a set of sessions was logged. Four sessions were created at various times through the experiment. Each session used a single VP and was CPU bound.

All four sessions were started with no particular guarantee, but with access to best-effort time. Sessions B, C and D were created at 3 second intervals following the creation of session A, and requested 10%, 20% and 30% shares of the CPU respectively, each with a period of 4ms and with no access to extra time.

Figure 7.1 shows the percentage of the total CPU that each VP received over each scheduling period (i.e. between consecutive deadlines). The scheduling period for each VP is used since this is provides an accurate view of whether the contractual guarantee made to the session is being honoured. Since session A's VP is running without a guarantee, it is assigned a notional scheduling period of 100ms; hence, the trace shown is clearly coarser than the traces for the other sessions, whose VPs have fine-grained guarantees.

Also present in the system, but omitted from Figure 7.1 for clarity, are the System session and the Best-Effort session. Figure 7.2 shows separate breakdowns for all sessions in the system, including the System Session and the Best-Effort session.

As may be seen from the traces of time received, RCANE respects the CPU guarantees that have been given to the various sessions. In particular, sessions B, C and D each initially briefly consume varying amounts of CPU time (whilst running best-effort), but once they reserve guaranteed allocations without use of best-effort time, they can be seen to accurately receive the guarantees that they requested. Session A consumes any spare CPU time that

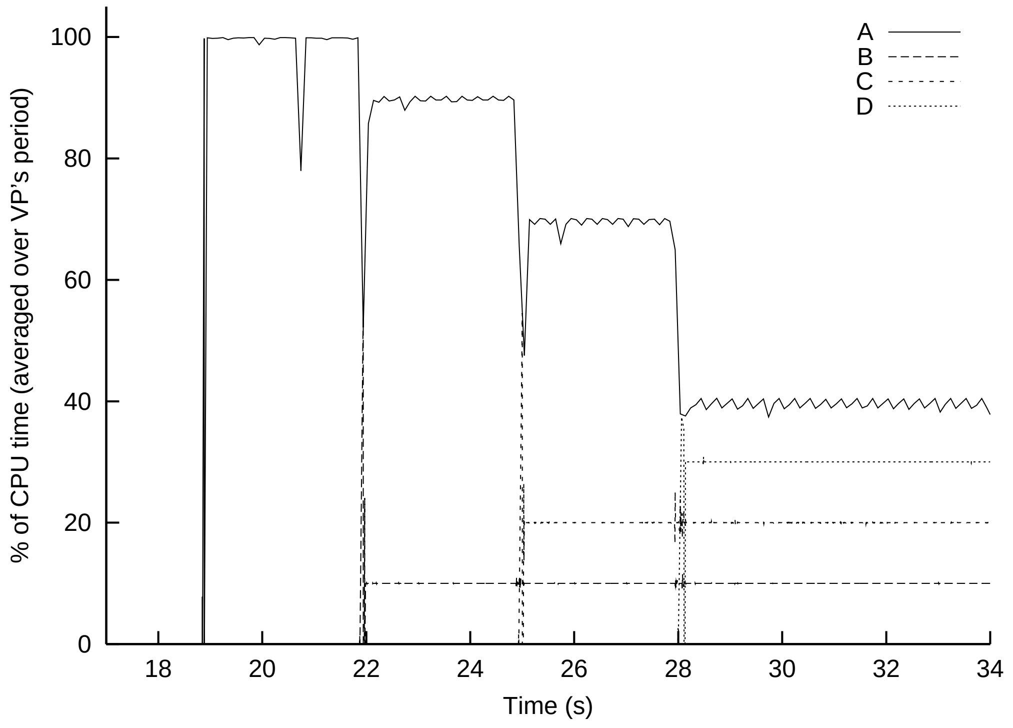


Figure 7.1: CPU consumption by a set of sessions

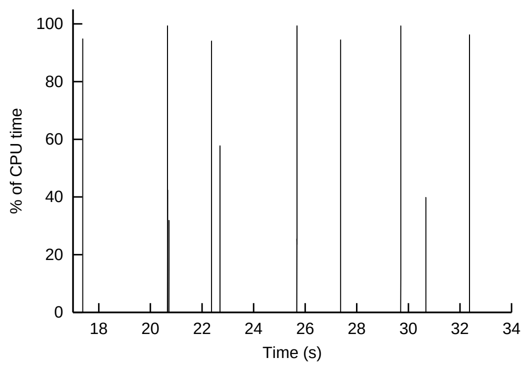
is available, and so initially receives 100% of the CPU – however, as each of the guaranteed sessions start in sequence, its share of the CPU is reduced.

The occasional spikes on the System session trace in Figure 7.2 (a) reflect periodic housekeeping activity (such as responding to routing information packets from other RCANE and PLAN boxes). The activity on the Best-Effort session trace in Figure 7.2 (b) reflects the work it performs in requesting the creation of new sessions. It can be seen that the sessions with guaranteed CPU time are virtually unaffected by this activity, whereas session A, which is running with no guarantee, experiences a loss of processor bandwidth.

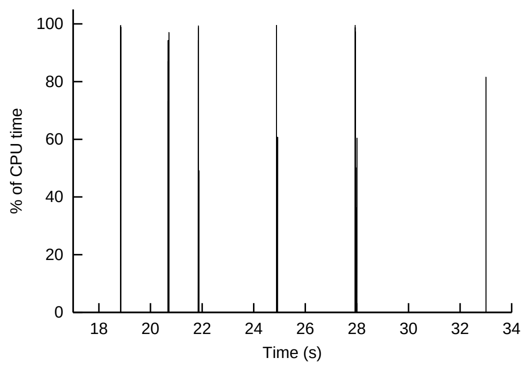
Thus it can be seen that sessions running on RCANE, by requesting a particular slice of CPU time over a particular period, may bound the latency with which they can process data received from other entities in the network, or calculate fresh data to be sent out to others.

Figure 7.3 shows the same experiment repeated, but with each session requesting access to best-effort time in addition to any guarantees that they might also request. It can be seen that the total time received by each session fluctuates considerably; this is due to the distribution of best-effort time being bursty at the scale of the sessions' scheduling periods. However, when the best-effort time received by each session is excluded from the traces (see Figure 7.4), it can be seen that the amounts of guaranteed time received by VPs belonging to sessions B, C and D still accurately match the reservations made by them.

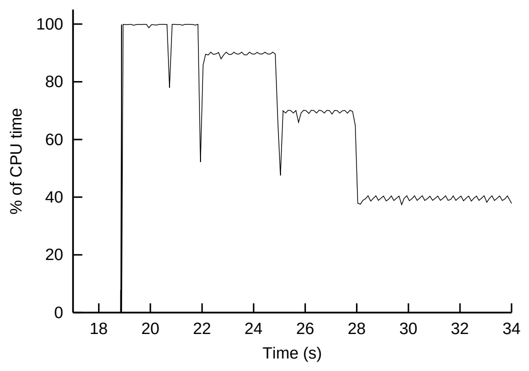
It was seen in Figure 7.3 that at a fine grain (at the granularity of the scheduling periods), the



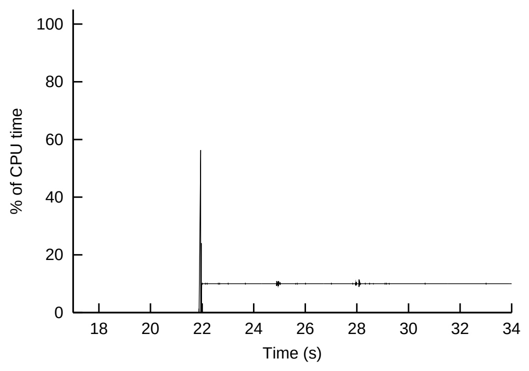
(a) System Session



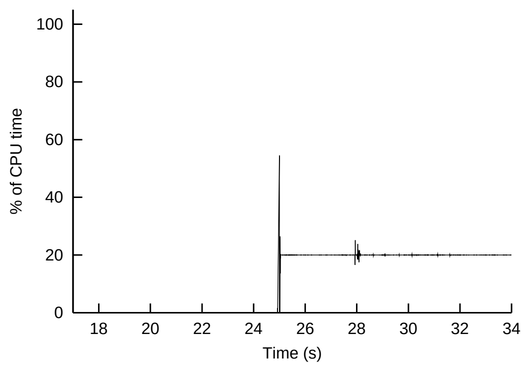
(b) Best-Effort Session



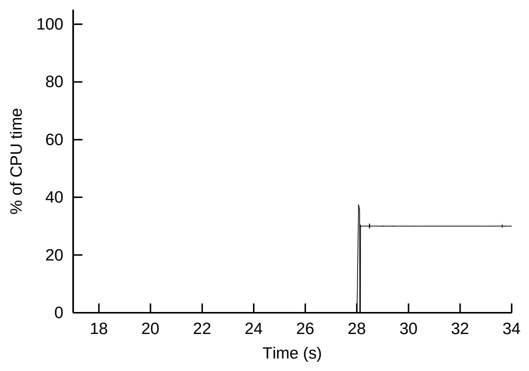
(c) Session A



(d) Session B



(e) Session C



(f) Session D

Figure 7.2: Total CPU consumption by a set of sessions (separated views)

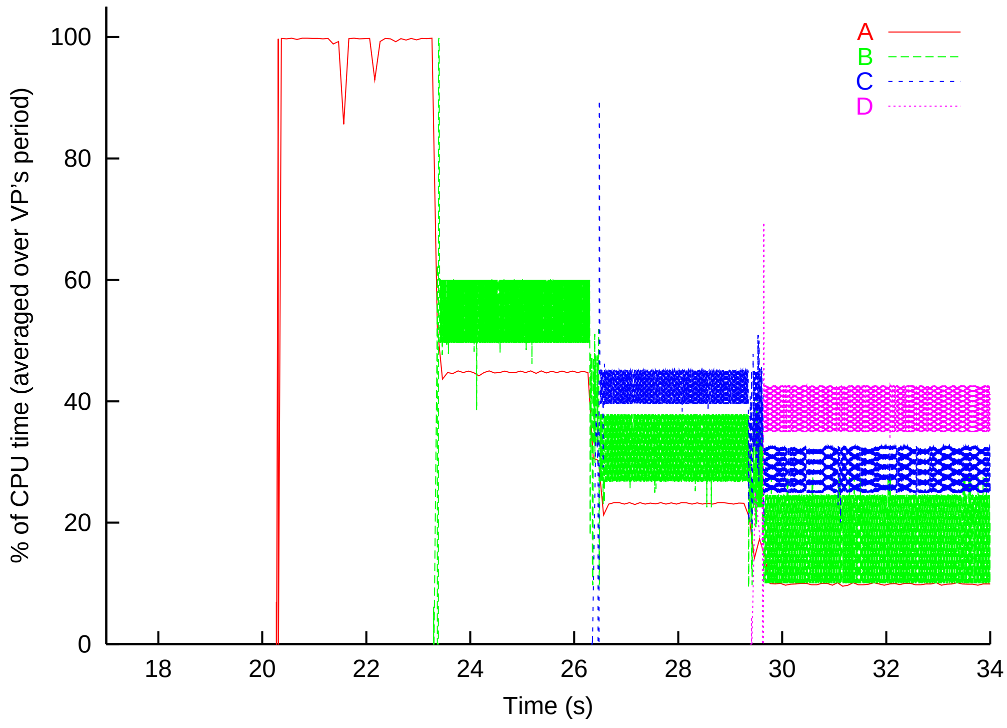


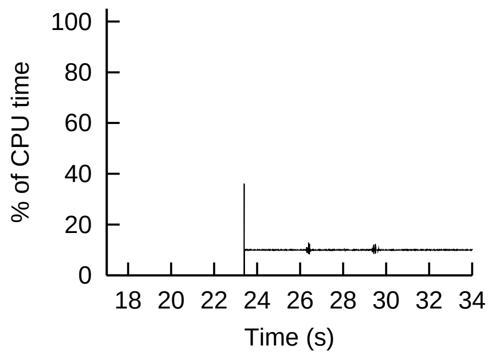
Figure 7.3: Total CPU consumption by sessions using best-effort time

amount of best-effort time received could vary substantially. Figure 7.5 shows the cumulative best-effort time received by each domain, recorded at the end of each scheduling period for each VP. It can be seen that at each point in time, the slopes of all running sessions are equal, showing that over longer periods of time the scheduler is fairly allocating any slack time in the system. As more sessions are added to the experiment, the amount of best-effort CPU time received by each session (and hence the slope of the line on the cumulative trace) decreases. As discussed in Section 6.3.1, in a complete implementation of Rcane, more sophisticated schedulers – with features such as sharing out slack time in proportion to the amount of guaranteed time reserved by each VP – may be used.

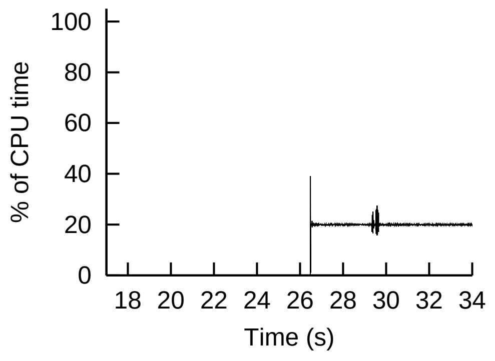
7.3 Service Performance

Rcane services are intended to provide lightweight communication between sessions. A major use of such communication is to access services provided by the system; alternatively it may be in order to access services provided by another session (e.g. enhanced routing tables) or to allow a particular execution environment to share state between multiple sessions. Straightforward data sharing is not possible due to the isolated nature of different heaps in Rcane, therefore the inter-session services need to have minimal overhead.

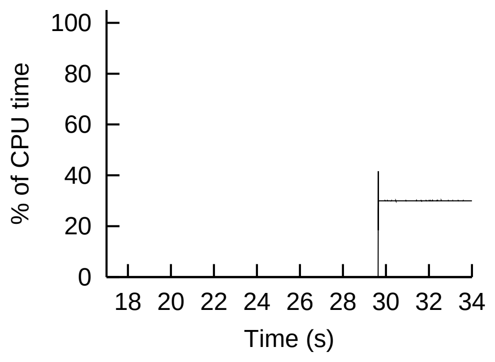
To measure the performance of the service mechanism, a series of test service invocations



(a) Session B



(b) Session C



(c) Session D

Figure 7.4: Guaranteed CPU time received by a set of sessions (separated views), when best-effort time is also being received by all application sessions.

were performed with different types and sizes of parameters. Also performed, for comparison, were a series of other non-service invocations. Each test was repeated one million times, and the average was taken. The large number of iterations used for the tests ensures that any amortised overheads due to garbage collection are accurately reflected. Such garbage collection runs will only occur every few thousand invocations, depending upon the size of the heap. The tests were repeated with substantially larger (several megabyte) heaps, which were fully garbage collected before commencing each test, to attempt to measure the overheads of copying without the overheads of garbage collection. However, these proved to be actually slightly slower than the tests that involved garbage collection. This is presumed to be due to the loss of cache locality when using a larger heap, and the cache pollution caused by the garbage collection runs between each invocation.

Table 7.1 and Figure 7.6 show the times taken for the various invocations. These invocations are:

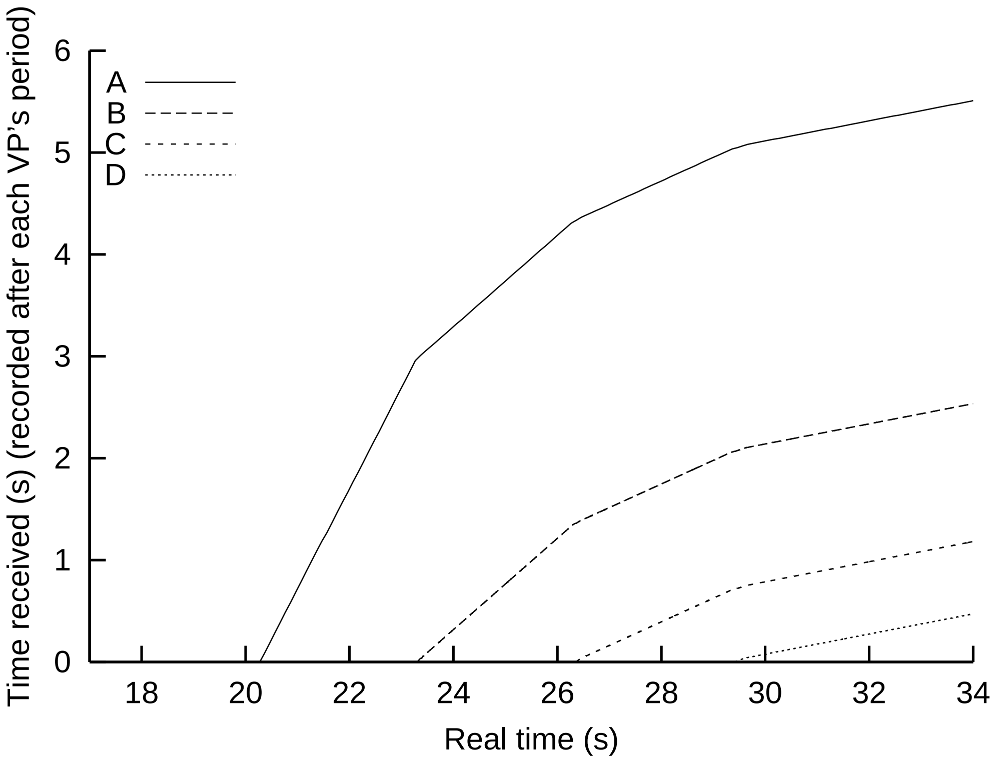


Figure 7.5: Cumulative best-effort time received

Runtime call: An invocation from the Caml virtual machine into the Runtime.

Local function: An invocation of a Caml closure from within the virtual machine.

Callback: An invocation of a Caml closure from within the Runtime.

Null service: A full inter-session service invocation with a single integer parameter[1].

Null service with timeout: A service invocation as above, but registering a timeout.

String service: An invocation of a service that takes a string, with string lengths varying over 0–100 bytes.

Array service: An invocation of a service that takes an array, with the array size varying over 0–100 elements (0–400 bytes).

List service: An invocation of a service that takes a linked list of integers, with the list length varying over 0–100 elements (0–1200 bytes).

The time taken by the runtime call, local function call and callback is intrinsic to the architecture of the OCaml virtual machine. It can be seen from Table 7.1 that a null service

[1]Due to the design of ML and the Caml VM, all functions take at least one argument. A void function is represented as a function that takes a value of type `unit`, of which there is only a single value, `()`, represented as an integer (zero) in the VM.

Invocation	Time (μs)
Runtime call	0.06
Local function	0.20
Callback	0.50
Null service call	1.50
Null service call with timeout	2.29

Table 7.1: Times for fixed-cost invocations

invocation is approximately six times slower than a local function call. An analysis of the time spent in a service invocation is given in Section 7.3.1.

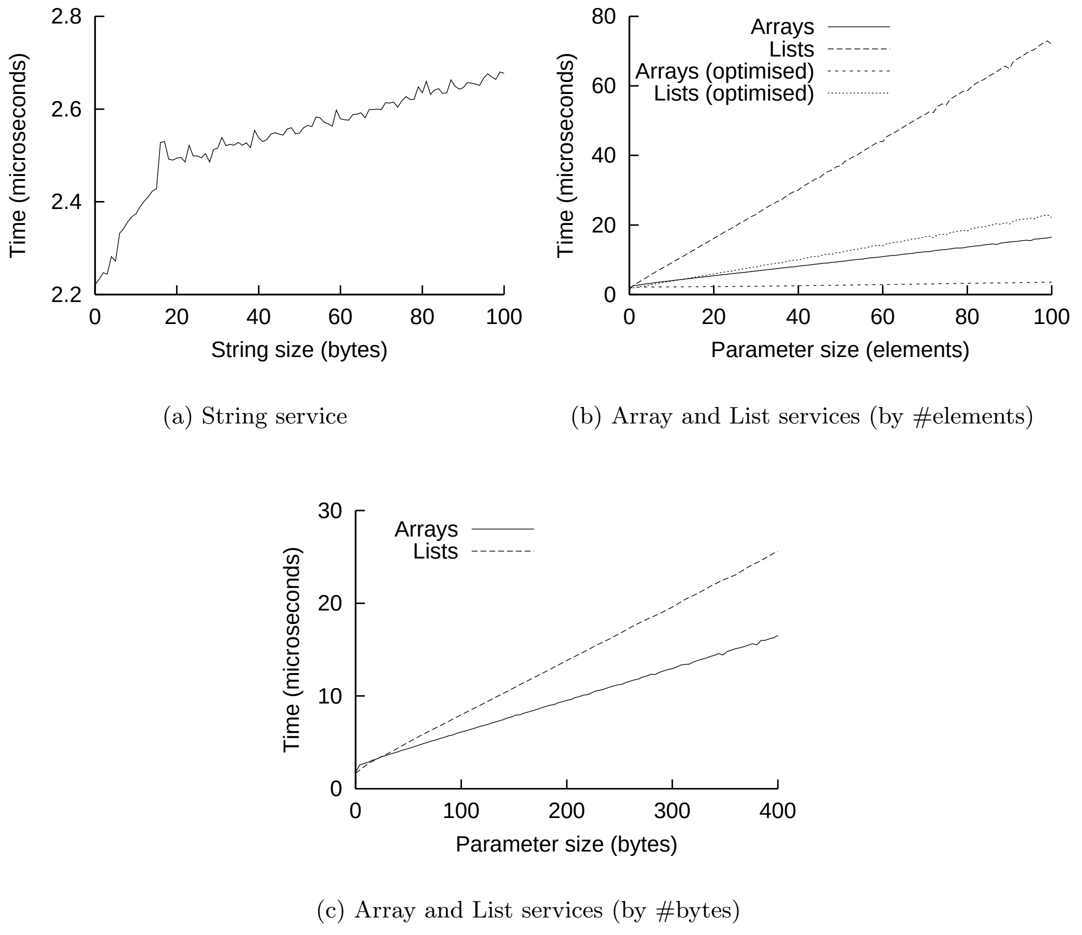
From Figures 7.6 (a), (b) & (c), it can be seen that in each case the cost increases roughly linearly with the size of the parameters. Figure 7.6 (b) shows the time taken for array and list services according to the number of elements being copied; Figure 7.6 (c) presents the same data according to the total amount of memory being copied – each list element consists of a two word cell plus a word of heap metadata.

In Figure 7.6 (b), optimised and unoptimised forms of the array and list invocations are shown. The string invocations and the unoptimised form of the array and list invocations represent three different modes of parameter copying for the generic copier:

- A string is a variable-sized object that may be copied in a single operation – the jagged nature of the trace is due to word-copy optimisations in the `memcpy()` function.
- An array is a variable-sized object whose elements must each be checked and copied individually.
- A list consists of a series of small fixed-sized objects each of whose elements must be checked and copied in turn.

As described in Section 6.7.3, there are opportunities to optimise the copying based on type information; e.g. a service parameter that is known to be an integer array can be copied in a single pass as with a string.

In many cases, the nature of ML's type system means that the type of an object cannot be determined until copy time; however, where possible RCANE uses type information to substitute optimised copying routines in place of the generic copier. It can be seen from Figure 7.6 (b) that the optimised versions of integer array and integer list service invocations are substantially faster than those using the generic marshaller. In the case of arrays, the time taken to copy even a 100 element array is insignificant when compared to the time taken for the remainder of the service invocation. For lists, the copying overhead is still substantial, due to the multiple allocations and the interactions required with the garbage collected heap; however, it can be seen to be significantly faster than with the generic copier.



(a) String service

(b) Array and List services (by #elements)

(c) Array and List services (by #bytes)

Figure 7.6: Service invocation times for varying numbers of parameters

7.3.1 Breakdown of time consumed in service invocation

In order to determine how much of the overhead of a service call is due to the mechanisms used by RCANE, and how much is due to the particular language used to implement the safe portions of RCANE, Figure 7.2 shows the breakdown of time amongst the various steps that make up an null service invocation.

This process of invoking a service is described in fuller detail in Section 6.7. Briefly, these steps consist of:

Local function invocation: Invoking the VM wrapper around the service handle.

Runtime invocation: Transferring control from the wrapper to the runtime.

Heap manipulation: Registering (and later deregistering) the calling thread with the server's heap, to ensure that any roots to server heap objects that are stored in the calling

Activity	Time		
	(μs)		%
	Cumulative	Incremental	
Local function invocation	0.24	0.24	15.2
Runtime invocation	0.34	0.10	6.3
Heap manipulation	0.60	0.26	16.5
Chain manipulation	0.77	0.17	10.8
Exception handling	0.90	0.13	8.2
Server callback	1.58	0.68	43.0
Total	1.58	1.58	100.0

Table 7.2: Breakdown of time spent in a service invocation

thread's stack are correctly traced.

Chain manipulation: Linking the calling thread into the list of callers for the service, for use in the event of server or client death.

Exception handling: Linking the runtime stack frame into the Caml exception chain, to ensure that any exceptions raised by the service invocation are handled in the runtime and appropriate recovery performed before returning to the client.

Server callback: Recursively invoking the OCaml virtual machine in the server's heap to call the actual service function.

Almost half the time taken by a service call is due to the costs of invoking the Caml virtual machine on the server's function. If the client specifies a timeout for the call, this adds an additional 0.7μs to the time required, representing an overhead of 45%. This cost consists of the time required to add and remove a timeout event for the service invocation.

By comparing Table 7.1 and Table 7.2, it may be seen that some of the steps take longer when executed as part of a service invocation than when executed on their own – this is presumed to be due to reduced cache locality.

7.3.2 Comparison with other IPC systems

Table 7.3 compares the performance of RCANE inter-session services with IPC primitives provided by other systems[2]

Nemesis Null IPC tests measure the average time for 100000 iterations of a full Nemesis IPC call; I/O Ping tests measure the time taken for an event to be passed over an I/O Channel from one user-level thread to another in a different domain, and back. These tests were performed for the case where both processes were in the same protection

[2]The figures in Table 7.3 for Windows NT, J-Kernel and L4 were taken from [Hawblitzel98].

IPC type	Hardware	Time μs
Windows NT RPC	PPro-200	109.0
Windows NT COM out-of-proc	PPro-200	99.0
Nemesis Null IPC (same protection domain)	PII-300	48.3
Nemesis I/O Ping (same protection domain)	PII-300	39.5
Nemesis Null IPC (different protection domain)	PII-300	69.8
Nemesis I/O Ping (different protection domain)	PII-300	56.8
Rcane Null service invocation	PII-300	1.5
J-Kernel Null LRMI (MS-VM)	PPro-200	2.2
J-Kernel Null LRMI (Sun-VM)	PPro-200	5.41
L4 Null IPC	P-133	1.8

Table 7.3: Time taken for a null invocation by various IPC systems

domain (thus removing the need to perform a full context switch between the processes) and also the case where each process ran in its own protection domain.

Windows NT RPC and COM tests measure a null invocation using two standard Windows NT IPC mechanisms.

J-Kernel Null LRMI (Local Remote Method Invocation) measures a call through a J-Kernel *Capability* [Hawblitzel98], running under a Microsoft JVM and a Sun JVM.

L4 Null IPC measures the time for an IPC in the L4 microkernel [Härtig97]

It may be seen from Table 7.3 that Rcane services are substantially faster than IPCs on Windows NT or Nemesis; they are comparable with IPCs over the J-Kernel. J-Kernel and Rcane both take a similar approach to protection, by employing multiple disjoint sets of data in a single address space, isolated through the use of type-safe code. An exact comparison of the efficacy of Rcane and J-Kernel is difficult, since while Rcane has more runtime support than the J-Kernel – which is implemented purely in Java (JIT-compiled where applicable) – the J-Kernel does not provide isolated heaps for each client. Thus Rcane has to perform additional work on a service invocation to link (and later remove) the client's thread into the server's heap.

The L4 microkernel has been optimised to provide fast IPCs between small processes through the use of non-overlapping memory segments, thus avoiding much of the overhead of address-space switching experienced by Nemesis and Windows NT IPCs. When the overhead due to the Caml virtual machine is discounted from the Rcane service invocation times, and accounting for the relative speed of the test equipment (300MHz vs. 133MHz) it can be seen that L4 and Rcane achieve comparable performance.

7.3.3 Abnormal Termination

In order to deal with arbitrary behaviour from user-supplied code (in particular, clients or servers exiting voluntarily or through resource exhaustion during a service invocation), Rcane services must be robust to abnormal termination situations. Four such scenarios were considered:

Server death: If a server session is destroyed or exits whilst a service invocation from a client is in progress, the client thread must be cleanly transferred back to the invoking session, and a `Revoked` exception raised.

Client death: If a client session or thread exits whilst in a service invocation to a server, the thread must be cleanly disconnected from the invocation chain for that server before the thread is destroyed. Garbage collection will ensure that any allocated data structures are released.

Client abort: This is similar to client death, with the difference that it is initiated by the client, and a `Service.Aborted` exception is raised in the client, rather than the thread being destroyed.

Client death with backup: In either of the above cases, when the server was executing within a critical region at the point when the abort/death occurred, and had registered a backup thread pool for the critical region's mutex the state of the client thread is transferred to the backup thread, allowing execution of the critical region to be completed cleanly.

Experiments were performed in which a client made service invocations to a server, and the different cases listed above were made to occur. In each case RCANE behaved as intended. This demonstrates that the good performance provided by RCANE services does not come at the expense of safe handling of (uncommon) abnormal termination cases.

7.3.4 Resource Backup

In Section 7.3.3 the thread backup mechanism of RCANE was evaluated to ensure that termination of a client thread within a server's critical section does not result in corruption of the data protected by that critical section.

As described in Section 5.6, since critical sections represent regions of mutual exclusion, it is important that a server can prevent a resource-poor or malicious client from spending too long in a critical section due to lack of CPU resources; otherwise crosstalk between clients may be experienced. Such control is afforded to the server through the use of the thread backup mechanism – by assigning CPU resources to the backup threads, server CPU resources may be used on behalf of the client only in those sections of code that might cause crosstalk, such as critical sections.

To demonstrate that this mechanism allows the server to provide an effective bound on the crosstalk experienced by a client, a simple server and a pair of clients were constructed. The bulk of the work performed by the server did not require access to a critical section, and hence could be carried out without a backup thread. Additionally, each call also updated some statistics about server usage; these statistics were protected through a critical section.

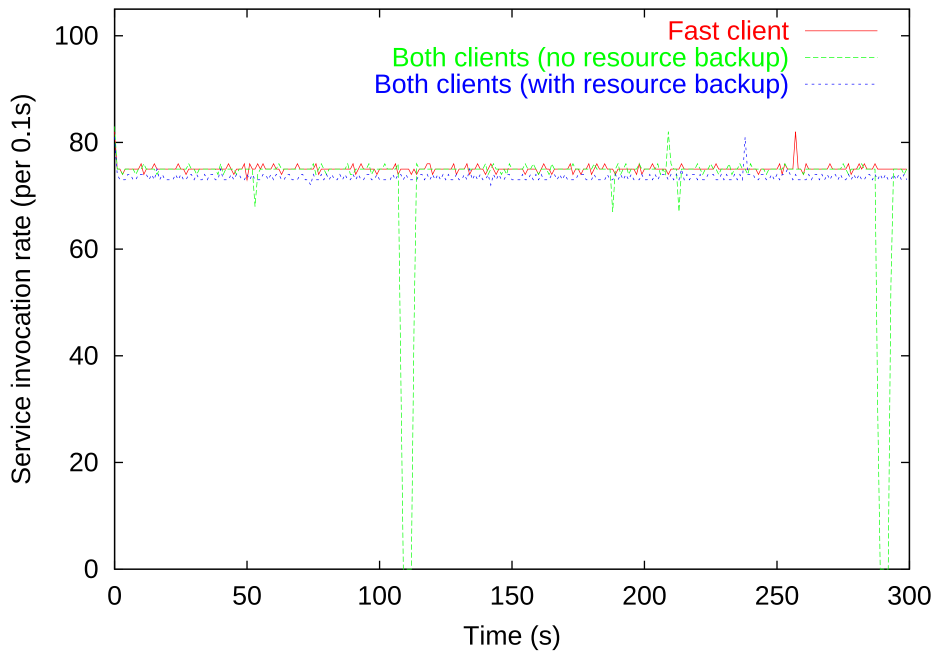


Figure 7.7: The effect of resource backup on crosstalk

Of the two clients, the first client (referred to as *fast*) received a 1ms CPU guarantee over a 10ms period, and the second (*slow*) received a 1ms CPU guarantee over a 500ms period. In order to simulate a heavily-loaded system, neither client was given access to best-effort CPU time.

Three experiments were conducted:

1. The fast client was run on its own.
2. Both clients were run simultaneously; no resource backup was employed by the server.
3. Both clients were run simultaneously; the server backed up the critical section with its own CPU resources.

In each case, the clients continuously invoked the services provided by the server, and the rate of service completion by the fast client was measured. Figure 7.7 shows the results for the three scenarios. It can be seen that when the fast client is executing alone, it achieves a constant service rate.

When the slow client is also executing, but without resource backup in the server, the fast client may be observed to occasionally experience severe crosstalk, for periods of up to 0.5s. This is due to the slow client exhausting its CPU slice whilst executing within the critical section. In this situation, the slow client does not receive any more CPU resources until the start of its next period (after 0.5s), and the fast client is unable to obtain the mutex for the critical section since it is held by the slow client; thus it cannot make use of its own guaranteed CPU resources. This is similar to but distinct from the problem of priority inversion [Lampson80] – since Rcane's CPU scheduler uses the notion of guarantees rather than priorities, there is not the same potential for deadlock as in a priority inversion.

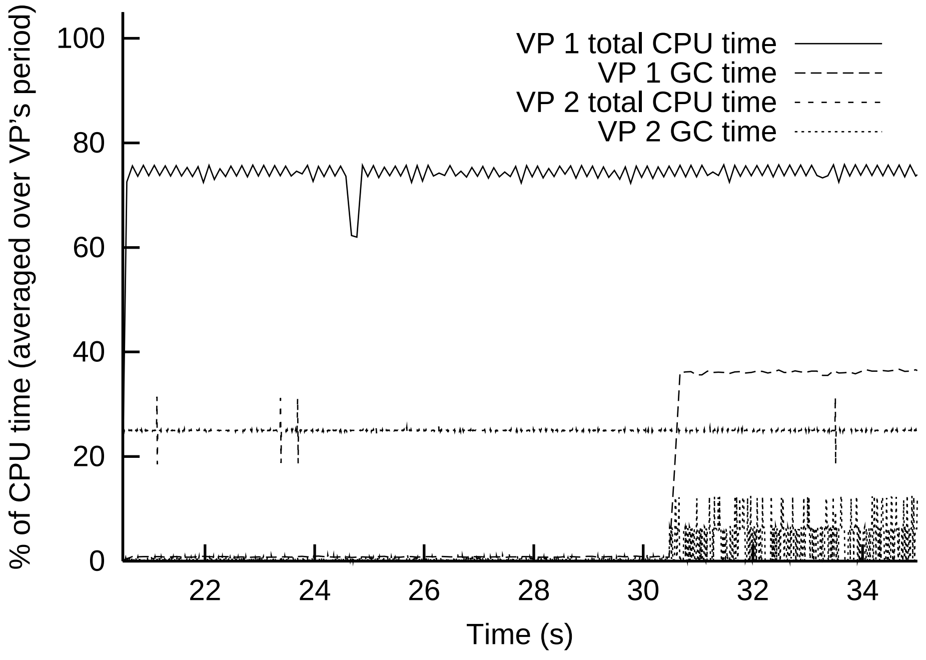
In the third case, when the server provides resource backup to the critical section, it can be seen that the fast client does not suffer any such crosstalk – on the occasions when the CPU resources of the slow client are exhausted while it is holding the critical section lock, the resources of the server are used to ensure its progress until it has exited the critical section.

Rcane's resource backup mechanism can thus be seen as an effective tool for supporting the client-resourced nature of Rcane's inter-session services, whilst still allowing a server to bound the level of crosstalk experienced by clients when accessing shared data within critical sections.

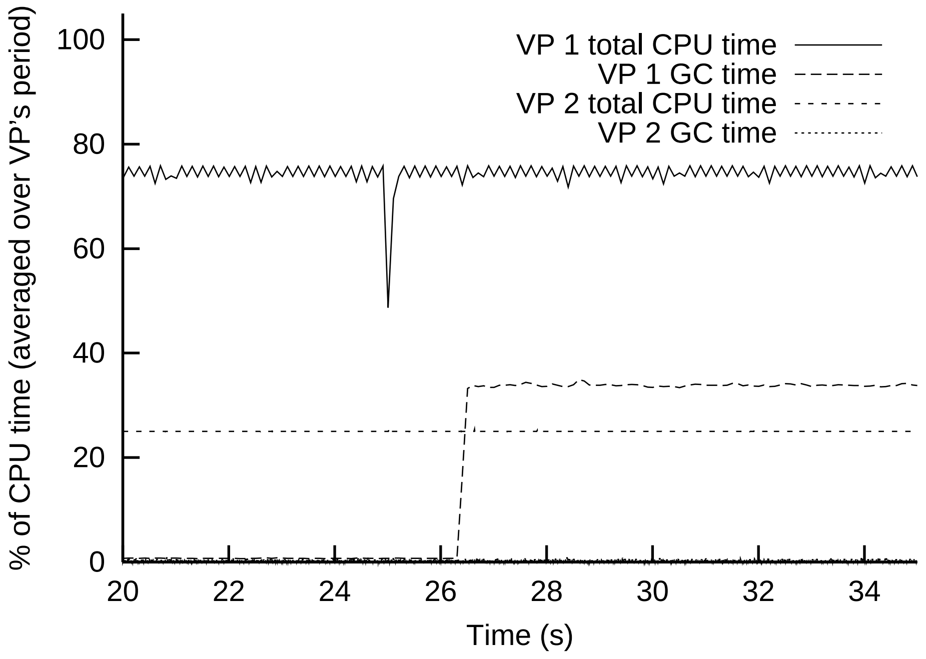
7.4 Garbage Collection Isolation

Rcane runs each session in a separate heap in order to eliminate crosstalk caused by garbage collection – a session should not be inconvenienced due to the allocation behaviour of other sessions. To demonstrate the utility of this approach, two scenarios were considered. In (a), VPs A and B are running in the same session. Initially both are generating small amounts of garbage. After a period of time, A 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 7.8 shows the outcome of these scenarios. In (a), both VPs are initially doing small amounts of GC work. A is running best-effort, whereas B has a guarantee of 1ms in each 4ms period. When A switches to generating large 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, B also ends up doing an irregular but substantial amount of GC work. Although B has its own independent CPU guarantee, at times critical GC activity (such as root tracing) is taking place when its thread is due to run; this GC work must be completed before normal execution can be resumed. In (b), B is unaffected by the extra GC activity caused by A, since it is running in a separate session and hence does not share its heap; thus B's own GC activity remains minimal.

This suggests that the decision to place each session in its own heap was a sound one. It should be noted that this represents a worst case scenario, in that session A was allocating very large amounts of memory with very little computation in between allocations, and thus stressing the garbage collector; however, since Rcane is designed to accept code from untrusted users, such byzantine behaviour may be seen from malicious or buggy clients. Isolating each client and accounting their resource usage directly is fairer and more reliable than attempting to identify this kind of behaviour based on heuristics or arbitrary limits.



(a) Single heap



(b) Separate heaps

Figure 7.8: Avoiding QoS crosstalk due to garbage collection

System	Time (ms)
RCANE	1.7
Nemesis	13.9
Linux (null)	0.4
Linux (quit-static)	0.7
Linux (quit-dynamic)	2.8
Linux (ocaml)	20.0
Linux (java)	394.0

Table 7.4: Time to create and destroy a process/session

7.5 Session Creation/Destruction

Until the full potential of programmable networks is widely realised, it is difficult to predict the typical lifetime of a session (representing a set of resource reservations) or the rate at which new sessions will be created. However, since maintaining an unused resource reservation on a node is likely to be expensive and wasteful, the architecture of RCANE should not, if practical, preclude the possibility of clients creating and destroying sessions over short timescales. Therefore the overhead of creating and destroying a session on RCANE needs to be low.

Table 7.4 shows the time taken to create and destroy a session in RCANE, and compares it with the time taken to create and destroy a domain in Nemesis and a variety of different processes in Linux (running on equivalent hardware). The various cases are as follows:

RCANE: A `wait4Last()` function was added to the `Session` interface, to permit a session to wait for its most recently created child session to exit, in a similar manner to `wait4()` in Unix systems. The Best-Effort session repeatedly creates a session and waits for it to exit. The child immediately exits.

Nemesis: A `WaitFor()` method was added to the `DomainMgr` interface to permit a session to wait for a given domain to exit. A benchmarking program repeatedly creates a child domain that exits as soon as the standard Nemesis environment has been set up.

Linux (null): The parent repeatedly calls `fork()` followed by `wait4()`, and the child immediately calls `exit()` (without executing a new image).

Linux (quit-static): As above, but the child instead `exec()`s a statically linked program that immediately calls `exit()`.

Linux (quit-dynamic): As above, but the child binary is dynamically linked.

Linux (ocaml): As above, but the child binary is a Caml virtual machine, executing a bytecode file that immediately quits.

Linux (java): As above, but the child binary is a Java virtual machine.

Activity		Time (%)
Creation	Creating Runtime structures	9.3
	Creating Core housekeeping structures	10.7
	Linking of minimal environment	27.3
	Dynamic linking of Caml/RCANE environment	30.6
Destruction	Revocation of services	16.6
	Releasing Runtime structures	0.8
	Releasing Core housekeeping structures	4.7

Table 7.5: Breakdown of time spent creating/destroying a session

In each case, the figure in Table 7.4 represents the total time between successive creations in many thousands of ongoing tests. In particular, the test for RCANE was repeated one million times with no increase in the memory usage of the RCANE domain.

The session under test in RCANE performs very little work itself, but the environment in which it runs is dynamically linked together at session creation time; a full Caml standard library is available, along with a full set of services for accessing the `Session` interface[3] that provides the main interface between the system and client sessions.

It can be seen from Table 7.4 that RCANE compares well to the dynamically-linked binary under Linux, particularly since in the current implementation of RCANE, all layers apart from the Runtime are run as interpreted Caml bytecode.

The overhead of creating and destroying a session in RCANE is also substantially lower than the cost of running a separate Caml (or Java) virtual machine for each client, or creating a separate Nemesis process.

A breakdown of the time consumed in creating and destroying a session is given in Table 7.5. This process is described in more detail in Section 6.8.

7.6 Network Performance

Without access to the network, an RCANE node is something of a white elephant; the CPU and memory resources are likely to be available more conveniently and more cheaply on the end-user's own node. It is the combination of these flexible programmable resources with the location within the network or close to other resources that enables the benefits of a programmable network to be realised. In particular, similarly to the need for timely access to CPU resources recognised in Section 7.2, timely access to the network is required to permit users to gain the latency benefits from code mobility, or to process streams of multimedia data.

Therefore, this section examines the network performance of the RCANE system, in terms of the quality of service guarantees that it provides to clients executing on the system.

[3]As partially described in Figure 5.2

7.6.1 Network Transmission

Figure 7.9 shows a trace of network output from three sessions, each attempting to transmit continuously.

- Session D has no guaranteed bandwidth.
- Session E has a transmission scheduling period of 6ms, during which it receives 2ms worth of link bandwidth. This 33% share equates to 33Mb/s on the 100Mb/s link used for this experiment.
- Session F has a 6ms transmission scheduling period, and a dynamically changing guarantee (see below).

Session F starts with a guarantee of 25%. After about 12s, it requests 45%, thus reducing the best-effort bandwidth received by D. After another 2s, it requests 65%. Now the link is saturated and there is no best-effort transmission time available for D. 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.

The apparent noisiness of the traces shown for the sessions with guaranteed resources is explained by Figure 7.10, in which a 1 second period from the same experiment is shown magnified. Sessions E and F are both seen to be oscillating around the actual value that they had been guaranteed. This is due to the quantisation effects caused by the use of large (1500 byte) packets and small transmission periods.

The network transmission guarantee given to session E, 33Mb/s guaranteed over a 6ms period, is equivalent to $16.6\dot{6}$ packets per scheduling period. Since it is not possible to preempt access to the network whilst a packet is in the process of being transmitted, sessions alternately overrun their guarantee in one period, and then receive correspondingly less in the following period, such that when the traces are averaged over two or more periods, the oscillation becomes insignificant. In the case of session E, it manages to transmit either 16 or 17 packets in each period.

This effect could be reduced through the use of a networking technology that uses a much smaller maximum packet/cell size. For example, ATM uses fixed-sized cells of 48 bytes (plus 5 bytes of header information), which could therefore allow scheduling decisions to be made on a finer granularity.

7.6.2 Network Reception

On a shared, unscheduled medium such as Ethernet it is not straightforward to provide guarantees on the number of packets received at a node for a particular client. However, the level of guaranteed buffering and CPU time that a particular client receives will affect the amount of incoming data on a network stream that actually reaches the client. To demonstrate

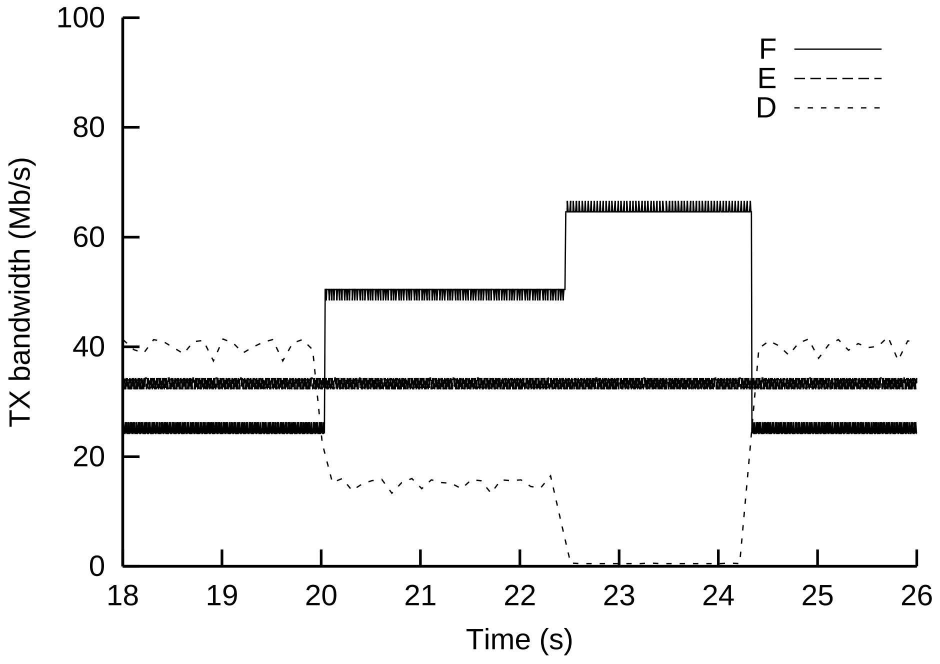


Figure 7.9: Network transmission rates

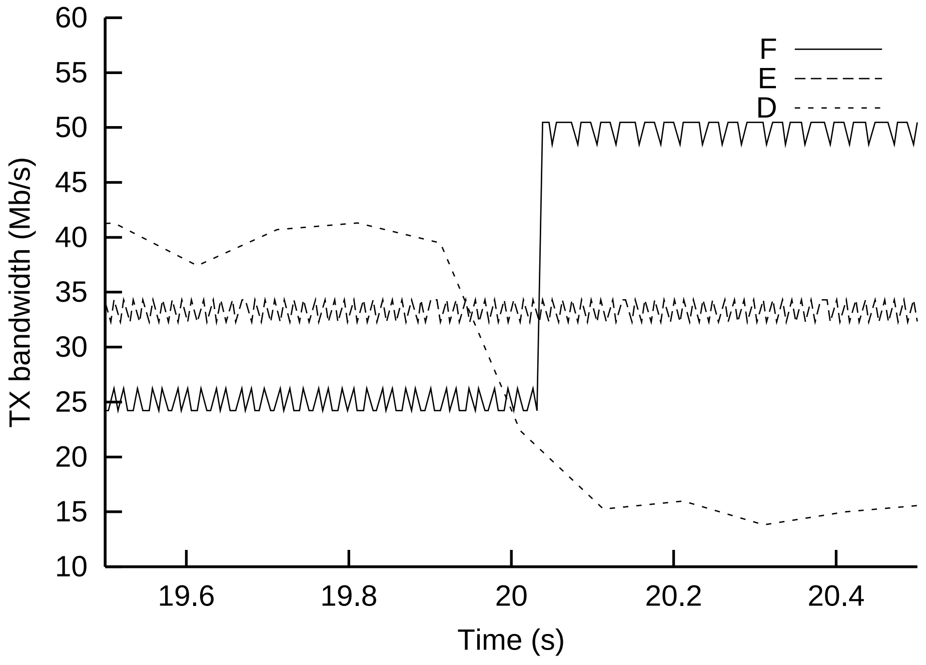


Figure 7.10: Network transmission rates (magnified view)

this, the bandwidth of data that could be processed by a session running on an Rcane node was measured.

A session on a neighbouring node (belonging to the same principal, and bearing the same session ID) transmitted approximately 97Mb/s across the intermediate link. A CPU bound session with a 40% guarantee and a 10ms period is also running on the receiving node. The buffer space reserved by the receiving session was varied between 1.5KB and 88KB; it was guaranteed 10% of the CPU, with a scheduling period varying between 1ms and 20ms. More details of the experimental setup, including the method used to launch the various sessions, are given in Section 8.2.3. 10% of the CPU had been observed to be more than sufficient to process the entire incoming data stream on an otherwise unloaded node.

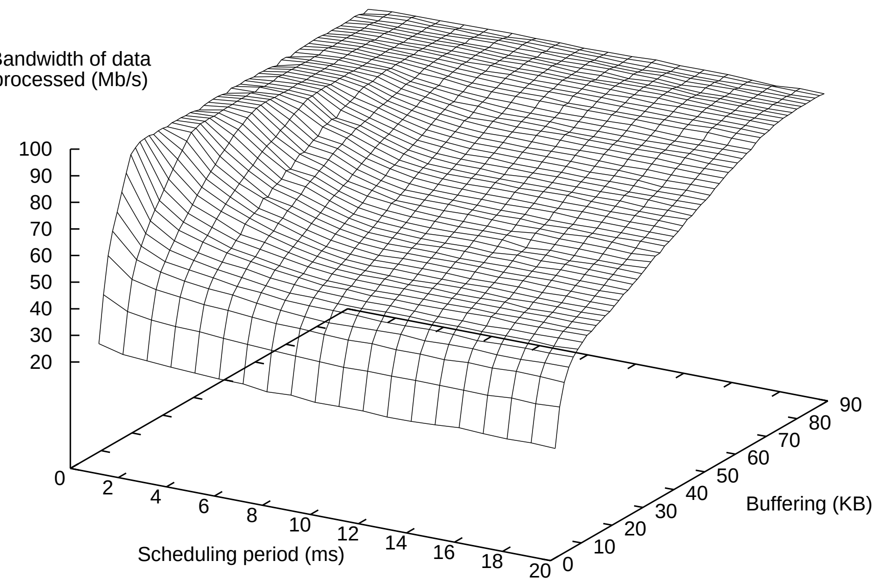


Figure 7.11: Network receive bandwidth as a function of CPU scheduling period and buffering

Figure 7.11 shows how the level of buffering and CPU scheduling period affect the amount of data that can be processed. It can be seen that when the session runs with a short scheduling period, it is scheduled sufficiently frequently to process all the incoming data even when its buffering is relatively low. Similarly, when the session has large amounts of buffering, it is able to process all the data even if its scheduling period is long.

However, the CPU-bound session has a guaranteed slice of 4ms every 10ms; as the receiving session's scheduling period increases, it finds itself interrupted for longer periods of time, and thus for modest levels of buffering the sustained receive bandwidth falls – the receiving session loses access to the CPU for long enough stretches of time that its buffers are filled up, resulting in incoming packets being dropped in the network device driver. As the receiving session's period increases significantly beyond the 4ms slice that is being received by the CPU-bound session, no further reduction in processed bandwidth occurs.

It may also be seen that at very low levels of buffering, the amount of data that can be processed drops off rapidly due to the inherent latency in the Nemesis I/O channels between the network device driver and RCANE.

The ability to make guarantees to clients about the level of resources reserved for them is essential to allow clients to make tradeoffs (such as a longer scheduling period[4] but a larger amount of buffering) so that they may meet their deadlines with maximum economy.

7.7 Summary

This chapter has presented an experimental evaluation of the key features of the RCANE architecture in terms of performance and quality of service. The evaluation was structured into five main sections.

The scheduler was examined, and was shown to provide effective guarantees of processor bandwidth to sessions running in RCANE. Such guarantees are vital to gain the latency improvements associated with code mobility; they are also required for jitter-free processing of streams of data within a network.

The RCANE inter-session service architecture was examined. RCANE services were shown to be efficient, and comparable with some of the fastest current IPC primitives. The type-based marshalling optimisations performed by RCANE were shown to provide substantial improvements over a marshalling strategy that takes no account of type information. The breakdown of a service invocation showed that a substantial portion of the overhead in a service invocation is inherent in the Caml virtual machine used by this implementation. These results demonstrated that RCANE's services provide an effective lightweight communication mechanism between otherwise isolated sessions.

Such isolation was seen to be effective at ensuring sessions received their CPU guarantess rather than being interrupted by garbage collection activity due to other sessions; the approach taken by RCANE of providing multiple independent heaps provided good guarantees even in the presence of heavy garbage collection activity caused by other sessions' pathological use of memory.

RCANE's sessions were shown to be a lightweight alternative to processes in a general-purpose operating system; in particular, creating and destroying an RCANE session was seen to be substantially faster than instantiating and destroying a full virtual machine for each client. Such a property is likely to be required of a node in a programmable network, as clients may wish to cancel their reservations when not needed, or transfer their reservations around the network. When running untrusted code, and in situations in which users' sessions may be terminated due to lack of resources, RCANE must be capable of correctly handling abnormal termination situations without error. The different abnormal cases were considered

[4]Since a shorter scheduling period requires more frequent reschedules and hence more scheduler overhead, it would be likely that an active node would charge more for a reduced scheduling period.

experimentally; in each case the correct behaviour was observed.

Timely access to the network is essential if end-users are to make effective use of a programmable network platform such as RCANE. RCANE was seen to provide effective resource guarantees for both network transmission (bandwidth) and network reception (buffering). The ability of a session to trade off its CPU scheduling requirements against its network buffering requirements was demonstrated.

The experimental results presented in this chapter have shown that at a system level, RCANE provides the resource isolation and guarantees required for an effective programmable network platform. The next chapter investigates the use of RCANE as an active network platform.

Chapter 8

RCANE as an Active Network Platform

The previous two chapters presented an implementation of RCANE and evaluated key aspects of the system. It was demonstrated that RCANE provides a platform with good resource isolation and guarantees. In order to properly act as a programmable network node, RCANE must be be sufficiently flexible to accommodate the multiple different styles of programming that active networks, open signalling and other programmable network applications require.

This chapter examines the use of RCANE as the basis for an Active Network node, with support for multiple different execution environments:

- The PLANet environment;
- The ANTS environment;
- Active Reliable Multicast.

8.1 Active Networks

In an active network, some or all of the nodes in the network may have their functionality programmatically extended by end-users of the network. The environment in which such user-supplied code may execute is known as an *Execution Environment* (EE). The EE defines the language or languages in which the code should be written, and the services that are available to clients of the EE. It has been suggested that IP itself constitutes a simple EE, using a language with one principal (implicit) service, `route-and-forward-packet()`, and a collection of less commonly used services (ICMP messages).

In earlier active network research projects, each node has generally supported only a single EE. Two research efforts are underway to make it more straightforward to use multiple EEs on a single node:

- The Active Network Backbone (ABone) [ABone] permits multiple EEs to run on a single Unix node as separate processes. The Active Network Daemon [SRI99] receives packets in the Active Network Encapsulation Protocol (ANEP) [Alexander97a] and demultiplexes the packets to the relevant EEs.
- The NodeOS [Peterson00a] aims to provide a low-level operating-system style interface suitable for implementing active network EEs.

In both of these efforts, the main resource principal is the EE – an EE is a program that controls resources on behalf of multiple end-users of the network. Whilst this model may facilitate the straightforward development of an EE, it makes it difficult for end-users to accurately control their own resources, particularly if they are interacting with multiple different EEs simultaneously.

The approach taken by RCANE is that the *session* is the resource principal, and that EEs should effectively be regarded as library code[1] that a session may instantiate to provide the desired environment. Thus a client with simple needs may interact directly with the raw interfaces provided by RCANE, whereas a client with more complex needs may instantiate one or more different EEs to provide a suitable environment services. In this way, the *client* has full control over its resources, and can allocate them to its tasks as it wishes, rather than being forced into a particular strategy provided by the EE.

A further advantage of this approach is that EE writers do not need to take resource management specific issues into account – instead, they simply provide hooks through to RCANE's resource management, and leave the details to RCANE.

It should be noted that there is nothing to prevent an end-user or a service provider from creating an RCANE session that acts as a traditional EE supplying active network services to other end-users; in this situation, the EE could still make use of RCANE's resource management facilities to help apportion resources between the different users of the EE. However, some features of RCANE – such as the heap isolation – are not applicable when isolating multiple activities within a single session; furthermore, the users of such an EE would then be forced to follow the resource allocation and programmability features of the EE.

The next two sections describe the implementation of two active network EEs over RCANE – PLANet and ANTS. These were chosen for several reasons:

- They are currently the two most popular EEs in use in the Active Networking community.
- They represent very different computation models: PLANet supports programs written in an extremely restricted language and carried with each packet, whereas ANTS supports demand-loaded protocol code in a general-purpose language.
- They are both complex EEs requiring a high level of flexibility in their underlying platform.

[1]The view that an EE is effectively a library available to applications is also finding favour in discussions [Peterson00b] regarding the NodeOS project.

In each case, an overview of the EE is given, followed by a discussion of the suitability of Rcane as a platform for supporting that EE.

8.2 PLANet

8.2.1 Overview of PLANet

PLANet [Hicks99c] is an EE that supports the execution of programs written in the Packet Language for Active Networks (PLAN) [Hicks98]. PLAN is functional language reminiscent of ML, but with very restricted functionality – in particular, unbounded recursion and looping are forbidden. PLAN programs construct complex behaviour by invoking *services* – routines written in a less restricted language, which have full access to the resources on the node. Such services are privileged code, and hence cannot generally be loaded by untrusted users.

A fundamental service provided as a primitive by PLAN is `OnRemote()` – this causes a remote evaluation of a PLAN function on a different node. Such an evaluation is implemented by sending a PLAN packet to the remote node. The body of the packet contains any PLAN code required for the evaluation – the function may either be specified by a name to be looked up at the remote host, or may be sent with the packet – along with any parameters that the function needs. Some of these parameters may be for control purposes, and others may represent data being transferred by the packet.

Other higher-level services provided by a typical PLANet node include:

- Fragmentation and reassembly of packets;
- Reliable delivery;
- Compression;
- Checksumming.

These services all make use of the concept of a *chunk* [Moore99b] – a first-class PLAN object representing a delayed function call. When a remote evaluation is made, the chunk is marshalled into a packet and sent to the remote node. Such marshalling may also be performed by services that wish to provide layered protocols.

For example, suppose we wish to execute the `test()` function on a remote node, passing a parameter that is larger than the MTU of the intervening links. The chunk representing this remote evaluation would be specified in PLAN as `|test(bigArgument)|` – the vertical bars indicate that this is a chunk, rather than a direct function call[2].

Attempting to call `OnRemote()` with this chunk to evaluate the test function on the remote node would result in the raising of an `MTUExceeded` exception. Consequently the packet must

[2] In the latest versions of PLAN this would actually be written as `|test|(bigArgument)` due to syntax changes.

be fragmented before being sent, and then reassembled at the remote node. The fragmentation process is shown graphically in Figure 8.1 and described below.

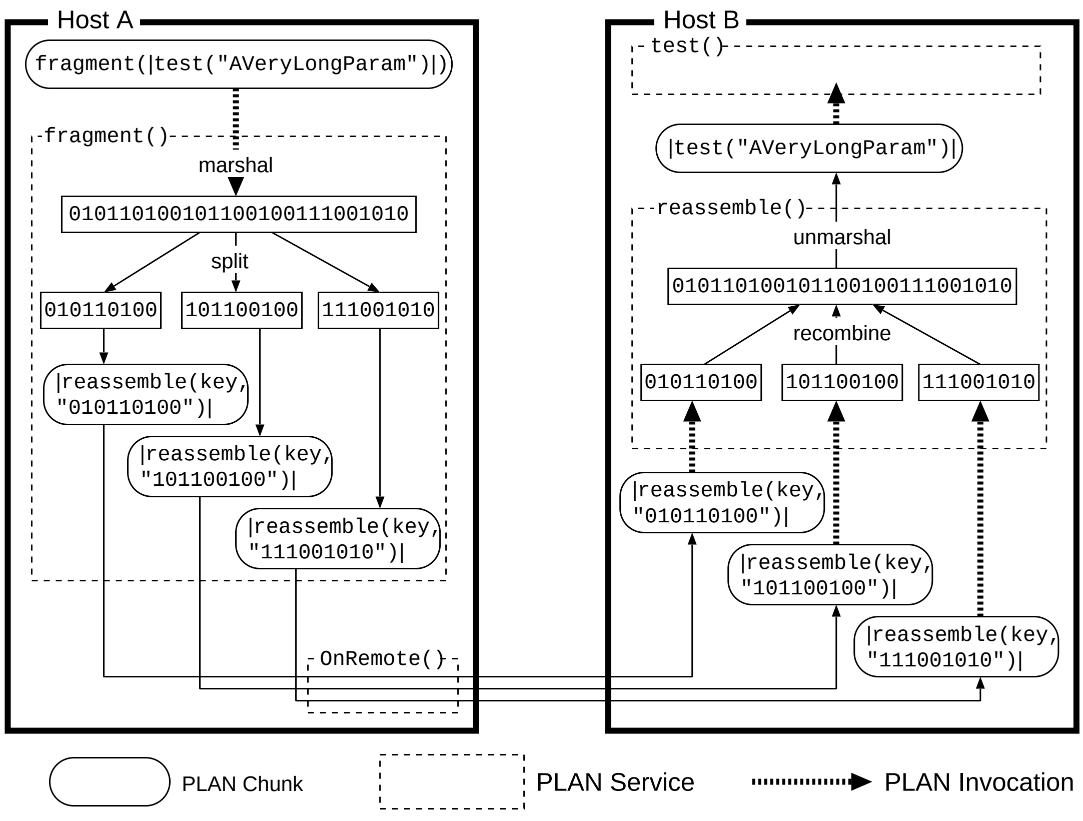


Figure 8.1: Fragmentation in PLAN – an example of the use of chunks

The `fragment()` service takes a chunk, marshals it into a buffer, and splits this buffer into fragments smaller than the MTU, taking into account the overhead introduced by fragmentation. Each of these fragments is wrapped into a chunk that, when evaluated, passes the fragment as a parameter to an invocation of the `reassemble()` service; an additional parameter is a key distinguishing between fragments from different packets generated by a single client. The `fragment()` service returns these smaller chunks to the program, which may now call `OnRemote()` without fear of exceeding the MTU limit. When these chunks are evaluated at the far end, each call to `reassemble()` stores a fragment of the original chunk in a table keyed on packet ID – once all the original fragments have been received, they can be stitched together to form a larger buffer. This buffer is then unmarshalled to recreate the original chunk, which is in turn evaluated to cause the desired invocation of `test()`.

Other services may be invoked to transform chunks in a similar way, so as to provide arbitrary encapsulation and layering of protocols. For example, an invocation of the chunk `fragment(checksum(|test(arg)|))` returns a list of fragments that, when reassembled into a single chunk, may be error-checked before being evaluated. Conversely, the invocation `checksum(fragment(|test(arg)|))` returns a list of fragments that will each be individually error-checked at the remote node before being reassmbled into the original chunk.

PLANet contains a very simple form of resource control – when a packet is sent into the network, it is associated with a *resource bound* value (RB). Each time a packet is routed by a PLANet router, its RB value is decremented. The RB is analagous to a hop-count or time-to-live field in an IP packet, with two extensions: a unit of RB is consumed on any recursive invocation; and when creating a new packet, the "parent" packet may choose how much of its RB to transfer to the new packet. The concept of the RB ensures that no PLAN packet can consume indefinite amounts of bandwidth or CPU time without connivance from PLAN services, which are trusted to manipulate packets' RB values only in safe ways. It does not, however, give any guarantee of timely access to resources, nor place concrete limits on the resources that can be consumed.

8.2.2 RCANE Implementation of PLANet

Since several of the low-level networking interfaces (but not implementations) that RCANE presents to clients were originally derived from interfaces used by PLANet, integrating PLANet with RCANE was relatively straightforward. PLANet has been the primary EE running over RCANE for large portions of the development, and has been used in experiments as a vehicle for transferring code to the RCANE box and for performing dumps of instrumentation traces, ARP-style host resolution, and routing services.

The lower-levels of the PLANet networking code were removed, since similar functionality – with the addition of resource isolation properties – is provided by RCANE. The main PLAN packet handler routine is now passed as a channel handler to the RCANE networking interfaces. Several portions of PLAN had to be reorganised to take into account the fact that functions previously handled by the PLAN EE itself were now being provided by RCANE, in the System session.

Since RCANE allows clients to reserve resources – in particular, heap memory for storing PLAN function definitions – clients may specify a particular PLAN function to be attached directly to the handler for a particular RCANE flow or subflow. This provides the convenience and safety of PLAN programs but without the additional per-packet overhead of repeatedly transferring and unmarshalling the required PLAN code or performing a symbol table lookup to find a specified function. Such an extension is impractical without an environment such as RCANE in which client code and data resources can be associated with particular network connections.

PLANet running over RCANE interacts with unmodified versions of PLANet running on Unix boxes, demonstrating that it is not necessary for all nodes in a network to be running an implementation of RCANE in order to benefit from RCANE's resource control models.

The prototype implementation of RCANE uses PLANet to provide its basic network services, such as routing table updates, neighbour discovery, address resolution (analogous to ARP) and session creation.

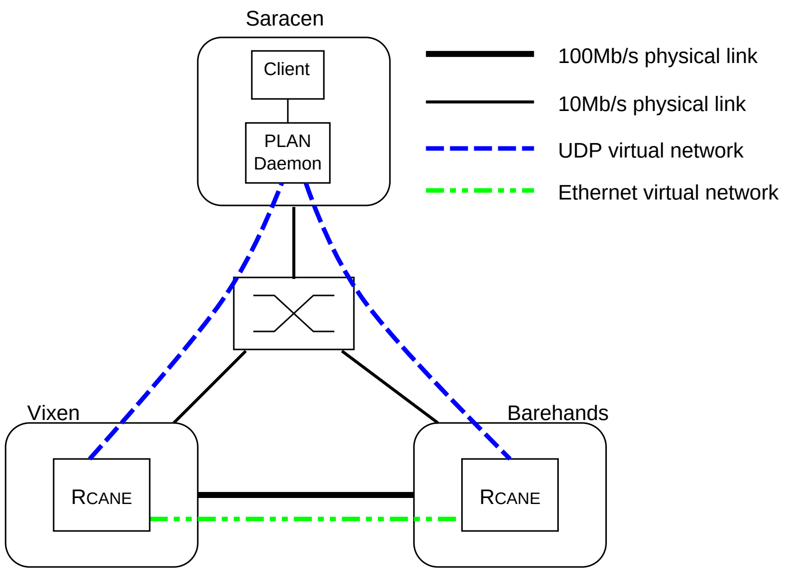


Figure 8.2: Experimental topology for network receive bandwidth testing

8.2.3 Experimental use of PLANet

To illustrate the use of PLANet over RCANE, and its interaction with standard PLAN nodes, Figure 8.2 shows the setup used by the network receive experiment described in Section 7.6.2.

Vixen and Barehands are RCANE nodes, and Saracen is a Linux node running a standard PLAN daemon. All three machines are connected via switched Ethernet; separate PLANet UDP virtual networks connect Saracen to each one of the RCANE nodes. A private 100Mb/s Ethernet connects Vixen and Barehands, over which is run a PLANet Ethernet virtual network.

A PLAN client running on Saracen connects to the PLAN daemon and sends a PLAN packet. This packet is evaluated within the daemon, and proceeds to send remote invocation packets to Vixen and Barehands. In each case, these packets create a session at the receiving node for the client. Each packet also contains the contents of a bytecode module to be loaded into the newly created session. The module loaded at Barehands continuously sends packets across the 100MB/s private link between Vixen and Saracen. The module loaded at Vixen requests guarantees for CPU time and network access, and then processes incoming packets. It monitors how its processing rate varies when it requests different levels of network buffering and CPU scheduling period.

An unrelated session is also running on Vixen; it is started at node initialisation, and represents contention for the CPU from other clients of the node. See Section 7.6.2 for details of the results of this experiment.

8.3 ANTS

8.3.1 Overview of ANTS

The Active Node Transfer System (ANTS) [Wetherall98] uses a demand-loaded approach to load Java code on to a network node. Each packet (or *capsule*) contains a type identifier that specifies the Java class that should be used to process the payload of the packet. The type identifier is typically the MD5 [Rivest92] checksum of the Java class file – since the behaviour of the processing algorithm is defined by the code, which is in turn used to generate the checksum, it is impossible to register a malicious protocol that will receive packets not intended for it[3].

When a packet is received at a node and its type identifier has been unmarshalled, ANTS looks in a code cache to obtain the relevant Java class; if this class is found, it is used to process the packet, otherwise a request is sent to the upstream node to ask for the relevant Java code. Since ANTS has just received the packet from this upstream node, it is very likely that the upstream node has a copy of the relevant code in its code cache. In this case the upstream node forwards a copy of the code for the packet and any related packet types belonging to its *Protocol* (an ANTS *Protocol* represents a collection of related packet types that may be used by a particular communications protocol), and the downstream ANTS node loads this code and uses the appropriate class to process the packet. In the event that the upstream node also has no copy of the appropriate class, the packet is dropped.

This model of code propagation allows protocol definitions to be "pulled" through the network behind the first packet of a particular protocol. Thus, over time, those nodes being used by flows from a given protocol will tend to have the code for that protocol in their code cache.

8.3.2 RCANE implementation of ANTS

Since ANTS is written in Java, and the current implementation of RCANE has support only for OCaml bytecode, a straight port of ANTS to RCANE was not possible. Instead, a simple version of ANTS was written in Caml to run over the RCANE virtual network interfaces.

The structure of ANTS over RCANE is shown in Figure 8.3. Each OCaml module that implements a protocol defines a function to process packets from that protocol. The *code table* provides a mapping from a module digest (the MD5 checksum of a OCaml bytecode module) to the protocol processing function implemented by that module.

The `Ants.demux()` function acts as the handler for network packets for a particular client. A session creates an ANTS environment, and passes a reference to `Ants.demux()` to the RCANE networking layer. When RCANE passes a packet to `Ants.demux()`, the type identifier of the protocol type is retrieved from the packet, and the appropriate processing function is called.

[3]Theoretically, if one could generate a malicious protocol whose Java bytecodes checksummed to the same MD5 digest as the protocol being attacked, one could subvert the security of the ANTS model – however, this is regarded as an intractable problem.

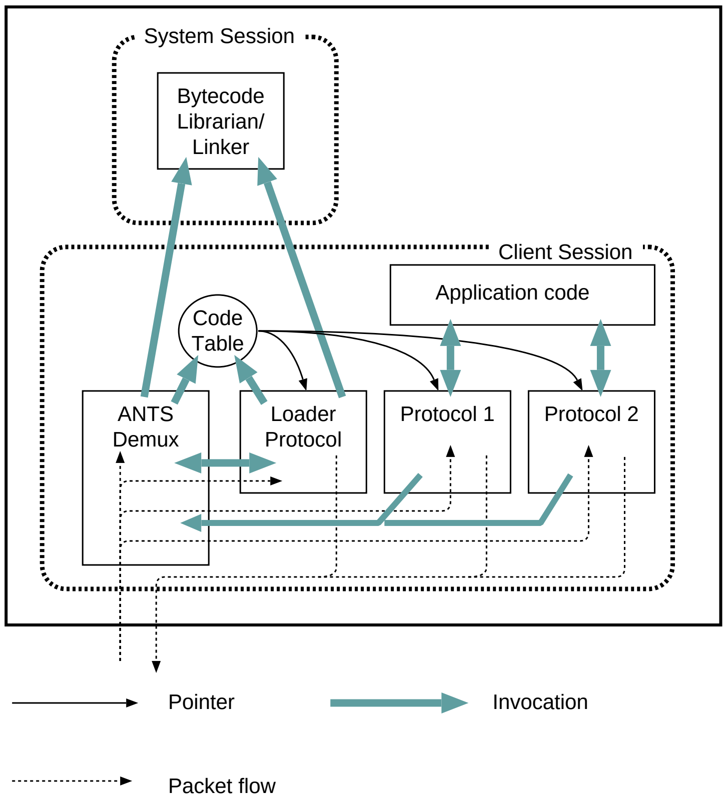


Figure 8.3: The structure of the ANTS implementation on RCANE

If no module has been registered for the specified protocol type:

1. ANTS asks RCANE whether a bytecode module with the appropriate digest has been loaded on the node by a different session. If such a module is available, ANTS binds to it, and registers its protocol processing function in the code table.
2. If the appropriate module is not found, the packet is added to a queue for that protocol type and (if necessary) a code request packet is sent to the upstream node. This request is itself an ANTS packet.
3. When a code reply packet containing the requested bytecode module is received from the upstream node, ANTS calls into RCANE to link the bytecode module and bind it into the session in which ANTS is running. The bytecode module is initialised, and its processing function is registered in the code table.

Although each session will have its own code table and instantiations of the protocol modules, the actual bytecode for each module is shared by all sessions, coordinated by the bytecode librarian in the System session.

The protocol processing modules may pass a partially processed packet back into the ANTS demultiplexer, thus allowing protocol layering in a natural way. As an example, the problem of fragmentation, which was presented in Section 8.2.1, is now considered for the RCANE implementation of ANTS.

Figure 8.4 shows how a custom protocol could be layered over `Frag`, a fragmentation and reassembly protocol when running on RCANE. When the first packet of the stream is sent out the following steps occur, assuming neither `Frag` nor `Custom` have been used recently at the destination:

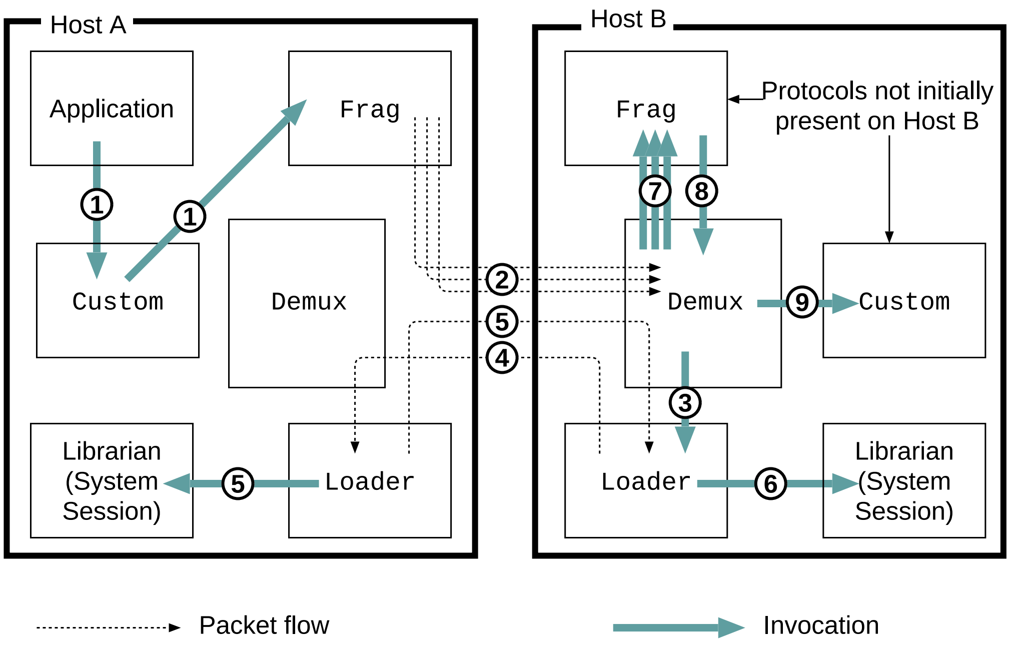

Host A
Application
Frag
Custom
Demux
Librarian (System Session)
Loader
Host B
Frag
Protocols not initially present on Host B
Demux
Custom
Loader
Librarian (System Session)
Packet flow
Invocation

Figure 8.4: Fragmentation in ANTS

1. The application calls `Custom`, which calls `Frag`, which fragments the packet.
2. `Frag` sends the fragments to B.
3. `Demux` on the destination looks up `Frag` and fails to find it, and hence calls the `Loader`.
4. `Loader` on B sends a request packet for `Frag`.
5. `Loader` on A queries the code table and/or the librarian and sends a reply packet containing bytecode for `Frag`.
6. `Loader` on B loads `Frag`.
7. `Demux` passes the fragments to `Frag`.
8. `Frag` reassembles the fragments into the original packet and passes it back to `Demux`.

   Steps 3–6 are repeated in order to load `Custom`.
9. `Demux` passes the original packet to `Custom`.

Future packets from the same stream reaching Host B would be directed directly to `Frag`, reassembled, and passed to `Custom` with no further network traffic.

This approach differs from the approach taken by PLANet in that users of a PLAN network are restricted to the services, such as fragmentation, installed by the node managers; therefore service innovations may occur over a longer time scale than users would like. With ANTS, end users can easily upgrade the services that they use within the network by simply sending packets bearing the module digest of the new service implementation. However, current versions of ANTS do not fully support untrusted code, due to a lack of resource control over user-supplied protocols [Wetherall99a].

These differences are largely removed when running over RCANE: both the ANTS and PLANet environments allow users to load their own code, although the out-of-band (PLANet) versus demand-loaded (ANTS) distinction still exists. The resource isolation provided by RCANE means that each user has a great deal of flexibility to customise their own environment, but cannot affect other users either in terms of resource starvation or by subversion of the EE by, for example, replacing a commonly used service in the PLAN symbol table.

8.4 Active Reliable Multicast

Much previous work has been done on implementing effective *reliable multicast*. Standard IP multicast [Deering89] provides IP delivery semantics to multiple recipients – i.e. delivery is unreliable, and there is no guarantee that the sender will be notified if the packet does not reach the destination. This lack of reliability has resulted in multicast mainly being used for video and audio applications, in which the loss of data merely degrades the subjective value of the application to the human user rather than causing inconsistencies between the various nodes in a distributed system. In contrast, reliable multicast aims to provide reliability on top of the multicast delivery, allowing the use of multicast for applications, such as a shared whiteboard, that are transferring data other than media streams. Additionally, reliable multicast may provide congestion control to reduce the load placed on the network by a particular flow during periods of overload, and causal delivery to prevent out-of-order delivery of data.

Many different reliable multicast solutions have been proposed, including SRM [Floyd97]; indeed, the IETF has felt it necessary to issue guidelines [Mankin98] to those developing new reliable multicast transports, giving criteria that all new proposals should meet and suggesting that:

> Due to the nature of the technical issues, a single commonly accepted technical solution that solves all the demands for reliable multicast is likely to be infeasible.

Protocols such as multicast have traditionally been implemented in network routers. In a field such as reliable multicast, however, where there is no clear agreement over the protocols to deploy, there is a risk that either different vendors will deploy incompatible solutions, or that many end-users will find that the deployed solutions are unsuitable for their requirements.

Consequently this appears to be an ideal domain for active networks – by providing a programmable interface on a network router, the vendors allow those implementing reliable multicast to deploy protocols and algorithms that suit their particular communication requirements, rather than restricting end-users to vendor-supplied protocols.

An example of such a protocol is the Active Reliable Multicast protocol (ARM) [Lehman98]. ARM was originally developed to run over ANTS, and to provide reliable one-to-many multicast semantics with no specific multicast support from the intervening network – the network needs only to provide programmability on a subset of its nodes.

Nodes running ARM maintain a cache of recent data. When a downstream ARM node notices that a packet has been dropped, it sends a NACK back up the multicast tree. When the next upstream ARM node receives the NACK, it will check its data cache for the requested packet; if the packet is present, it will be forwarded to the downstream node. If the packet is not present, the upstream node notes the fact that the downstream node needs a copy of the packet, and forwards the NACK to the next upstream node. NACKs for the same packet from multiple downstream nodes are coalesced, resulting in only a single NACK being passed to the next upstream node if the packet is not found in the cache.

When retransmissions are received, they are only forwarded to those nodes that have expressed an interest in that packet's retransmission, i.e. those nodes that have previously sent a NACK.

Thus ARM aims to reduce the bandwidth requirement for a reliable multicast stream in two ways:

- Upstream bandwidth is reduced since NACKs are coalesced. This helps prevent the phenomenon of *NACK implosion*, in which a packet lost at an early stage in the multicast tree causes a NACK from many (or all) receivers to be sent to the original sender.
- Downstream bandwidth is reduced since retransmissions are selectively targeted to the specific branches of the multicast tree that failed to receive the packet previously, and retransmissions from ARM caches in the network need not traverse the whole path from the sender to the receiver.

Furthermore, the latency for retransmissions is reduced; with no support in the network for caching and retransmissions, a NACK must travel all the way from the receiver to the sender, and the retransmission must travel from the sender to the receiver. This results in a minimum latency of one round-trip time to recover from a lost packet. Alternatively, the receivers themselves must coordinate to supply retransmissions to one another [Floyd97]. This has the drawback that multiple receivers may send the repair packet simultaneously; thus in order to prevent flooding the network with repair packets, each such receiver must wait a random amount of time before sending the repair, and aborting if it detects a repair from another receiver. With ARM caching in the network, the minimum recovery latency is a round-trip time from the receiver to the next upstream ARM node, rather than to the sender (as when the sender is responsible for repair packets), or a round-trip time to another receiver plus a random element (as when receivers are responsible for repair packets).

ARM nodes would be most valuably located near the ends of links that are expected to suffer loss; in this way the loss may be noticed and repaired at the earliest possible time, and the retransmission – which represents wasted use of the network links – will have the least distance to travel.

In [Lehman98], an ARM network is shown to have a recovery latency[4] of 0.2 RTT, compared with SRM [Floyd97], which approaches 1.4 RTT with 100 nodes in a multicast group. Furthermore, the majority of ARM's benefit, resulting in a recovery latency of 0.4 RTT, is achieved when only 30% of the ARM routers cached fresh packets (as opposed to repair packets, which were cached by all ARM routers). Also shown was that ARM could control NACK implosion to the same level as SRM when only "strategic" nodes – those with more than two outgoing links in the multicast tree – were running ARM, with other nodes running a standard multicast router; such strategic nodes typically constituted less than half the set of possible routers.

8.4.1 ARM on RCANE

The reduction in latency and repair bandwidth achieved by ARM is at the expense of increased memory (and, to a much lesser extent, CPU consumption) on the ARM nodes, since they need to cache data for later retransmission. In a programmable network without reservations, the deployment of a protocol such as ARM could be expected to lead to excessively large caches being maintained. Since the utility function of the cache is likely to monotonically increase with the size of the cache, there is little incentive for users to limit their usage of memory.

A possible solution would be to put a fixed limit on the amount of ARM caching at a particular node; however, this would have two drawbacks:

- It would not directly provide a way to favour important traffic, or traffic that would especially benefit from caching.
- Since the network is programmable, it would not prevent end-users from running their own ARM-like protocol that did not respect such limits.

Alternatively, this problem can be solved through the use of a programmable network platform such as Rcane, which is capable of providing, and accounting for, robust resource guarantees to clients.

The implementation of ARM described in [Lehman98] maintains a single ARM cache for all clients; under Rcane, a separate session would be created for each multicast stream, or group of related streams, at the nodes where the stream's users wished to employ active processing. ARM is treated as a shared library, which each client session may instantiate to perform active multicast processing.

[4]The recovery latency is defined as the time from a receiver detecting a packet loss to when it receives the first repair for that loss.

An implementation of ARM has been developed to run over RCANE. The low-memory callbacks described in Section 6.4.3 are used to inform ARM that it is approaching the limit of its reserved heap size. At this point older packets, along with older structures describing NACKs received and pending retransmissions, are recycled to reduce garbage collection overheads or released to be reused as a different type of object. ARM records the numbers of NACKs received, and the number of times that the packet requested by a NACK was not found in the cache. At intervals it checks the ratio of these, and if the ratio is too high, it deduces that the amount of caching it is using is insufficient to effectively deal with the loss on the downstream links, and requests a higher memory reservation from RCANE (assuming that the client is willing to pay for an additional memory reservation).

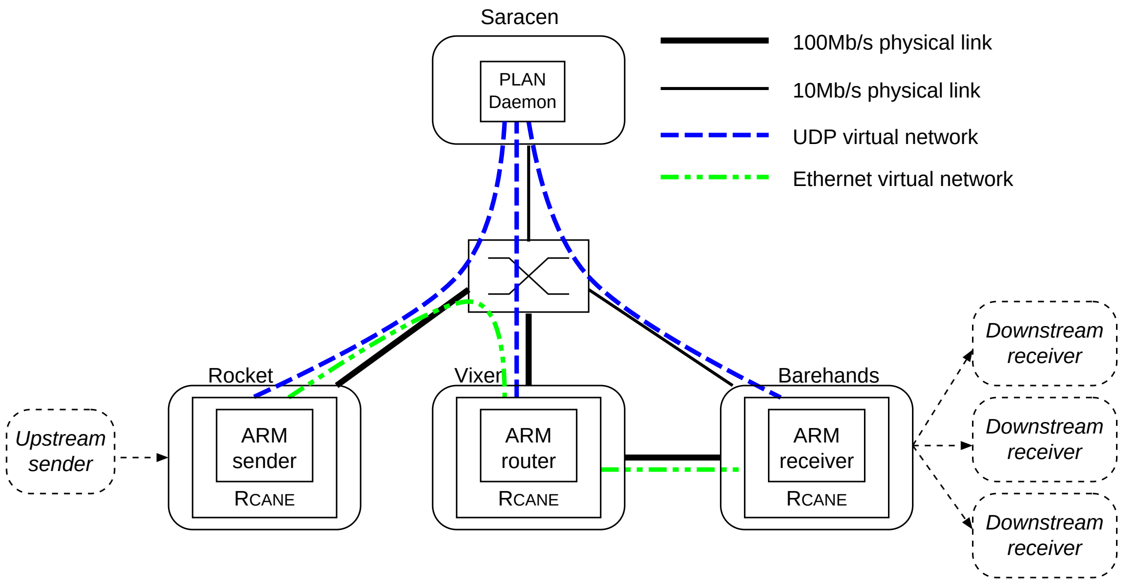


Figure 8.5: Experimental topology for ARM testing

Figure 8.5 shows the experimental setup used to demonstrate the use of ARM over RCANE. Rocket, Vixen and Barehands are each running RCANE and are connected by 100Mb/s Ethernet. Saracen is running a PLAN daemon and is used to control the RCANE machines. The link between Vixen and Rocket is assumed to be non-lossy. The link between Vixen and Barehands is artificially made very lossy, by dropping packets at Barehands with a given probability. Sessions on all three nodes are started by a client who wishes to transfer an ARM flow from upstream of Rocket to downstream of Barehands. Thus Rocket is acting as an ARM sender, Vixen as an ARM router and Barehands as an ARM receiver[5]. Initially, 25% of packets from Vixen to Barehands are dropped; after 90 seconds 50% are dropped. Vixen is configured to attempt to increase its reserved heap size if it discovers that it is servicing less than 90% of the NACKs from its cache over a 1 second period.

Figure 8.6 shows the memory usage of the ARM router on Vixen, along with the total number of packets sent, the number of NACKs received from Barehands and the number of cache

[5]Clearly in a real situation Rocket would be receiving from an upstream node, and Barehands would be sending to a set of downstream nodes; this was not practical due to lack of suitable networking infrastructure.

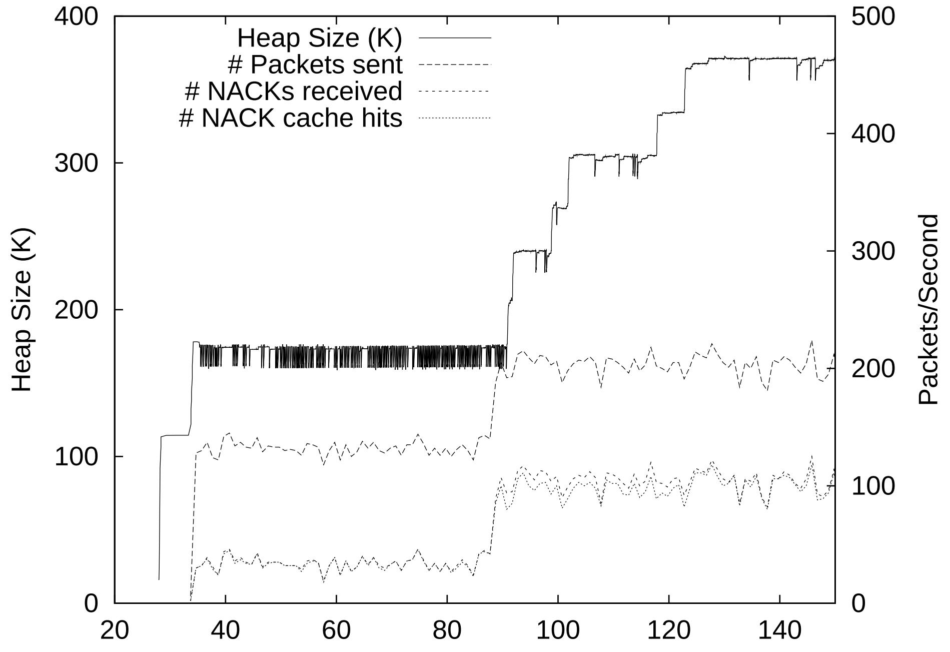


Figure 8.6: Memory usage, packet counts and cache hits for an ARM stream

"hits" (NACKs that could be served from the cache). It can be seen that the heap consumed by ARM quickly grows to approximately 180kB; at this level, Vixen is able to service over 90% of the NACKs that it receives from Vixen directly from its cache. After 90 seconds the loss on the link from Vixen to Barehands doubles to 50%. At this point, the number of NACKs received from Barehands increases significantly, and the percentage of NACKs serviced from the cache falls noticeably below 90%. Thus ARM increases its reserved heap size in an attempt to reduce the number of misses; the heap size grows to approximately 370kB over the course of 40 seconds, at which point the level of NACK hits reaches the target of 90%, and the heap size stabilises.

This demonstrates the ability of RCANE to constrain the amount of memory used by different clients on a node – an essential property of any practical programmable network platform.

8.5 Summary

This chapter has illustrated how RCANE provides support for multiple different active network and mobile code solutions. Two different active network models – PLANet, in which code is carried with each packet, and ANTS, in which code is loaded out-of-band – were implemented over RCANE.

Although PLANet has limited support for preventing packets in an active network from consuming unbounded amounts of bandwidth or CPU time, neither ANTS nor PLANet support the allocation of resources to clients, nor guarantee timely access to those resources.

When running over Rcane, both of these environments are treated as libraries available for use by applications; it thus suffices for access (or hooks) to Rcane's resource control interfaces to be provided in the environment.

The problem of reliable multicast was considered. Due to the difficulty of defining a single reliable multicast protocol that meets all users' requirements, this appeared to be an area in which active networking could be a practical solution; the benefits experienced through the use of ARM, an Active Reliable Multicast protocol, were discussed. However, without resource control between clients, such solutions could lead to excessive resource usage on nodes within the network. Rcane was seen to provide an effective platform for controlling such resources.

Chapter 9

Conclusions and Scope for Future Work

This dissertation has examined the requirement for resource control in open programmable networks. An architecture for supporting such resource control was presented, named RCANE – the Resource Controlled Active Node Environment. A sample implementation was examined and evaluated. This chapter summarises the work and its conclusions and makes suggestions for further areas of study.

9.1 Summary

Chapter 2 described the background to this research. It began by looking at "passive" end-to-end delivery networks, and their evolution into programmable networks, in which end-users may customise the processing in some or all of the nodes in a network. A review of the rationale for two of the main forms of programmable networks – active networks and open signalling architectures – was presented, followed by a taxonomy of different levels of programmability and a review of platforms currently used for programmable networks. In particular, the NodeOS – an attempt to define a common standard for programmable nodes in the same way that IP provides a common standard for datagram routers – was discussed. Related work in safe languages, resource control and extensibility was surveyed.

Chapter 3 examined the requirement for resource control within programmable networks. The high variance in traffic seen in current wide-area networks suggested that even in passive networks, the overheads of resource reservations may turn out to be cheaper than the costs of overprovisioning best-effort networks.

Open programmable networks were defined as networks in which some subset of the nodes provided programmable facilities accessible to all users of the network. The fact that users of programmable networks have much more flexibility, and that their resource usage cannot be conveniently limited through the use of bandwidth throttling, suggested that the requirement for resource usage was likely to be greater in programmable networks than in traditional passive networks.

Two different approaches to resource control, through proof properties and scheduling, were considered. The resources that required control, such as CPU cycles, network bandwidth and various types of memory, were discussed, along with possible approaches to controlling each.

In Chapter 4, the Nemesis Operating System was discussed. The features that it provides to facilitate resource control were examined.

The RCANE architecture was introduced in Chapter 5. The design principles underpinning the architecture were presented:

- Untrusted clients should not be able to adversely affect the operation of the node, thus the system should be partitioned into multiple layers of security.
- Resource consumption should be accountable to the entity causing it to occur, thus computation activities should, where possible, be partitioned on a per-client basis rather than through shared servers.
- Clients with guaranteed resources should not experience interference with those guarantees due to the actions of other clients.

The concept of a *session* was introduced as a means of accounting for resources consumed by a remote client, and the abstractions used by the architecture to support scheduling and accounting of various resources were presented. Each session can reserve one or more allocations of guaranteed CPU time, or *VP*s; a session's activities may be multiplexed over its VPs flexibly, according to its own priorities, through the use of threads and *thread pools. Events* allow the deferral of computations to a specified later time. *Channels* were presented as an abstraction of network streams; all processing associated with a channel is accounted to the owning client, and scheduled accordingly. RCANE gives each session its own isolated heap to prevent garbage collection crosstalk; to permit lightweight communication between sessions, a form of thread-migrating IPC called a *service* was introduced. The possible approaches to providing such services were examined in detail. A discussion of accounting and billing issues followed, accompanied by the related topic of detecting and preventing DoS attacks.

Chapter 6 described a prototype implementation of RCANE. The system was implemented over the Nemesis operating system and the Objective Caml bytecode interpreter. Nemesis was chosen for its good resource isolation properties; OCaml was chosen as a simple and flexible interpreter previously successfully used in active network research. Several deficiencies in these two systems that had to be remedied were discussed. Key aspects of the implementation were then presented. A modified form of EDF is used by RCANE to support effective guaranteed allocation of CPU time; Nemesis uses a similar form of scheduler to control network transmission bandwidth. To support RCANE's multiple heap architecture, the system must keep track of which threads have access to a particular heap, and ensure that garbage collection runs involve all appropriate threads. The implementation of inter-session services was described, along with details of optimisations to improve parameter marshalling, and mechanisms to handle abnormal termination.

An evaluation of the resource control facilities offered by RCANE was undertaken in Chapter 7. RCANE was shown to provide effective resource isolation between multiple non-cooperative clients, and to provide lightweight mechanisms for communication and session creation. Reservations of CPU and network output bandwidth were examined, and found to be respected; garbage collection interference between different sessions was found to be significantly reduced, compared to the situation in which multiple clients execute in the same garbage-collected heap.

Finally, the application of RCANE to existing programmable network execution environments (EEs) was considered in Chapter 8. Two active network environments – PLAN and ANTS – were ported to or rewritten for RCANE. An overview of each EE was presented, together with details of their RCANE implementations. The problem of packet fragmentation was presented for each EE to provide a comparison between the two approaches.

The advantages gained by running these EEs over RCANE were discussed. Neither EE previously had significant support for resource control; the use of RCANE as a platform means that such resource control need not be provided by the EE, but can be delegated through the provision of access to RCANE's interfaces. The Active Reliable Multicast protocol – designed to reduce recovery latency and bandwidth consumption in reliable multicast streams – was also implemented; the effectiveness of RCANE's memory consumption control was demonstrated.

9.2 Future Work

This section considers open problems in the area of resource control in programmable networks which could not be addressed in this dissertation due to lack of available time. Two main areas are considered: issues affecting a single node, and the integration of a single RCANE node into a larger programmable network.

9.2.1 Issues Affecting a Single Node

Several aspects of the RCANE architecture and prototype implementation with regard to resource control on an individual node in a programmable network have put forward issues that deserve further attention.

9.2.1.1 Persistent Storage

RCANE has not focused on providing access to persistent disk storage – although less transient than independent capsules, the computations considered have either been relatively stateless or sufficiently short-term not to require disk storage. However, it is envisaged[1] that some clients in a programmable network may have requirements for access to disk storage. An OS such as Nemesis, which already has support for scheduled disk access with low levels of

[1]The NodeOS Draft Specification, for example, contains an API providing POSIX-like file access.

inter-application crosstalk [Barham97], would provide a useful base for the development of effective resource-controlled mechanisms for persistent storage support.

9.2.1.2 Virtual Memory

For simplicity, this dissertation has not considered the use of paged virtual memory. This is not considered a great drawback, since an application that is paging to disc is likely to experience sufficient latency that any benefits of executing within the network – closer to the resources with which they wish to interact – are lost. However, as with persistent storage, some end-users with high latencies on to the network coupled with requirements for large amounts of state within the network may find it profitable to request an allocation of virtual memory in addition to physical memory.

Given support for persistent storage mentioned in the previous chapter, it may prove to be sufficient for such applications to manually "page" their important data structures in and out of files. A possible improvement on this scheme would be to utilise the Nemesis self-paging architecture [Hand99]. This would allow each session's heap to be backed by a *Stretch Driver* – an abstraction for controlling virtual memory. Each Stretch Driver would have a connection to a disk with a given level of QoS, and a certain set of physical pages to populate with virtual pages.

9.2.1.3 Inter-Session Services

The type-based optimisations for inter-session service argument copying described in Section 6.7.3 are currently rather limited. Further work needs to be done to provide more effective generation and sharing of marshalling routines. The mechanisms used by Nemesis' IPC marshalling system should prove to be adaptable to this task.

Also an open problem is the development of further and richer mechanisms to support the efficient aborting of services without compromising the state of servers.

9.2.1.4 Selection of Resource Levels

Calculating the level of resources that should be requested in an environment such as Rcane is not straightforward. Techniques such as progress-based feedback may be used to measure whether an application is satisfying the user's requirements, and purchase additional resources if not; alternatively task requirements, reservations and node resources may be specified in some generalised resource units.

For example, the Eclipse [Bruno99] project measured processor bandwidth reservations in terms of SPECint95 [SPEC95] units; for a given platform, these units could be calibrated by running the standard SPECint95 benchmarks. The NodeOS [Calvert98] has proposed logarithmic "standard RISC cycles" to serve a similar purpose.

A further possibility would be to measure applications' resources usage at run time to determine their actual resource requirements; such measurements could be used either to provide feedback to applications, allowing them to adapt their reservations based on their past usage, or to enable RCANE to provide a higher degree of multiplexing but with "probabalistic guarantees", as described in [Barham98].

9.2.1.5 Integrated Scheduling

Currently Nemesis supports only a single *scheduling domain* (`sdom`) per domain. A scheduling domain represents a guarantee from the CPU scheduler in the Nemesis kernel. Thus in order to provide different CPU guarantees to client sessions, RCANE must request a single guarantee with sufficiently low period and high slice to be able to meet its sessions' guarantees, and employ a user-level EDF VP scheduler above the kernel CPU scheduler, integrated with the pool/thread scheduler.

Current research [Stratford00] is investigating the separation of the `sdom` abstraction from the domain, and thereby allowing a single Nemesis domain to have multiple `sdoms`. Building on this work to provide each RCANE VP with a separate `sdom` would simplify the structure of RCANE and potentially reduce scheduling overheads.

9.2.2 Network Integration

The work presented in this dissertation has focused mainly on resource control at a single node, without full consideration for bandwidth reservations on shared links, or global resource policies. Such a network may be fully programmable, or mainly passive with programmable nodes placed at strategic locations.

9.2.2.1 Bandwidth scheduling

The current implementation of network scheduling in RCANE is concerned only with the local link bandwidth. To gain the full benefit from the resource reservations provided by RCANE, the network scheduling would need to be integrated with end-to-end bandwidth reservations. It would be necessary to provide the ability to reserve bandwidth and buffering for flows on routers and switches (either programmable or passive) between the node and the remote resources with which it was communicating.

9.2.2.2 Resource Transfer

More work is required to integrate RCANE's resource reservations with global resource allocation schemes. For example, it ought to be possible for a single session to transfer resources very straightforwardly from one node to another. The resource bound used in RCANE virtual network packets provides a very trivial form of such resource credit transfer; however, this has not been fully integrated with the actual resource reservation mechanisms.

9.2.2.3 Billing

The billing schemes proposed in Section 5.7 have not been implemented. Linking the accounting information gathered by Rcane to some concrete form of billing will be vital for the effective deployment of an open programmable network.

9.3 Conclusion

It is the thesis of this dissertation that resource control in active and programmable networks is essential if the flexibility offered by such networks is to be made open and available to general end-users of the network; and that such resource control can be practically accomplished. To this end, arguments have been presented to support the case for resource reservations in programmable networks. An architecture for providing such support has been exhibited, and its implementation and evaluation have been examined. It is concluded that the dissertation supports the thesis.

Appendix A

An Outline Design for an Objective Caml Verifier

In Chapter 6 the lack of bytecode verification for OCaml was discussed. This appendix presents a brief outline of a design for a verifier for the OCaml virtual machine. The design is not complete, and the verifier has not been implemented due to time constraints; however, this design supports the proposition made in Section 6.2.7 that the lack of a verifier for OCaml is not an insurmountable deficiency.

A.1 Introduction

In order to safely execute a bytecode module supplied by an untrusted remote client, it may be necessary to verify that the bytecode respects certain invariants; in the case of OCaml bytecode [Leroy97], such verification must check that the bytecode respects the Caml type system.

Verification involves tracing all possible paths through a module and ensuring that at each instruction, no type violation is committed. Such violations may be detected by showing the possible occurrence of one of several conditions:

- Using a pointer in an arithmetic context.
- Dereferencing an integer.
- Dereferencing an abstract object.
- Returning a value of the incorrect type from a function call.
- Passing a value of the incorrect type to a function call.
- Storing a value of the incorrect type in a structure.

- Modifying an immutable value.

These are all variations on the basic violation – i.e. using a value of one type in a context that expects a different type. The design presented in this chapter is for a a verifier that can identify such violations in OCaml byte code; since the verification is performed on bytecode rather than on source code, it may be used to verify OCaml virtual machine code generated from any source language, not just OCaml itself.

A.2 Concepts

In order to attempt to perform a violation of the type described in the previous section, the program must obtain the value from somewhere. In the case of the OCaml virtual machine, the set of addressable locations consists of:

- The *accumulator* – a single location that is used for most operations, including arithmetic, dereferencing, and the return value from a function call. The type of the value stored in the accumulator changes regularly at runtime.
- The *stack* – a per-thread set of values, used for temporary storage and parameter passing. Entries may be pushed on to or popped off the top of the stack at runtime. The type associated with a particular stack location may not altered without popping that location from the top of the stack and pushing a value of a different type.
- The *environment* – a per-closure structure that is initialised when the closure is created. The types of the fields in the environment are constant for a particular closure.
- The *global vector* – any values accessible through interfaces imported by the current module. The types of the contents of the global vector are constant for a particular module

Associated with each bytecode location is a tuple $(\mathcal{A}, \mathcal{S}, \mathcal{E}, \mathcal{G})$ consisting of the types of accumulator, stack, environment and global vector respectively. A simple fragment of Caml is given in Figure A.1; Figure A.2 shows how this tuple might be represented by the verifier.

The types of the stack and accumulator values are generally individual to each instruction; however, in the case of an instruction that does not modify the type of the accumulator nor perform any stack modifications – such as the `ADD` instructions in Figure A.2, the same stack/accumulator record may be shared with the next instruction. All stack/accumulator records for a given function share an environment record – in this case, the functions are simple enough that they have no environment, so there can be no valid access to the environment. All environment records in a module share the same global vector record.

Three different kinds of type may be stored in these tuples:

- *unqualified* types such as `int`, `int ref`, `float`, and `int` $\rightarrow$ `string`.

Function to increment its argument
`let inc x = x + 1`

Function to increment its argument's referent
`let incref x = x := !x + 1`

(a) Fragment of Caml code

PC	Instruction	Meaning
0	`ACC0`	$Acc \leftarrow Stack[0]$
1	`ADD 1`	$Acc \leftarrow Acc + 1$
3	`RETURN 1`	*Return*
5	`ACC0`	$Acc \leftarrow Stack[0]$
6	`GETFIELD0`	$Acc \leftarrow Acc[0]$
7	`ADD 1`	$Acc \leftarrow Acc + 1$
9	`PUSHACC1`	*Push Acc;* $Acc \leftarrow Stack[1]$
10	`SETFIELD0`	$Acc[0] \leftarrow Stack[0]$*; Pop*
11	`RETURN 1`	*Return*

(b) Compiled bytecode

Figure A.1: Sample Caml code fragment and its compilation in OCaml bytecode

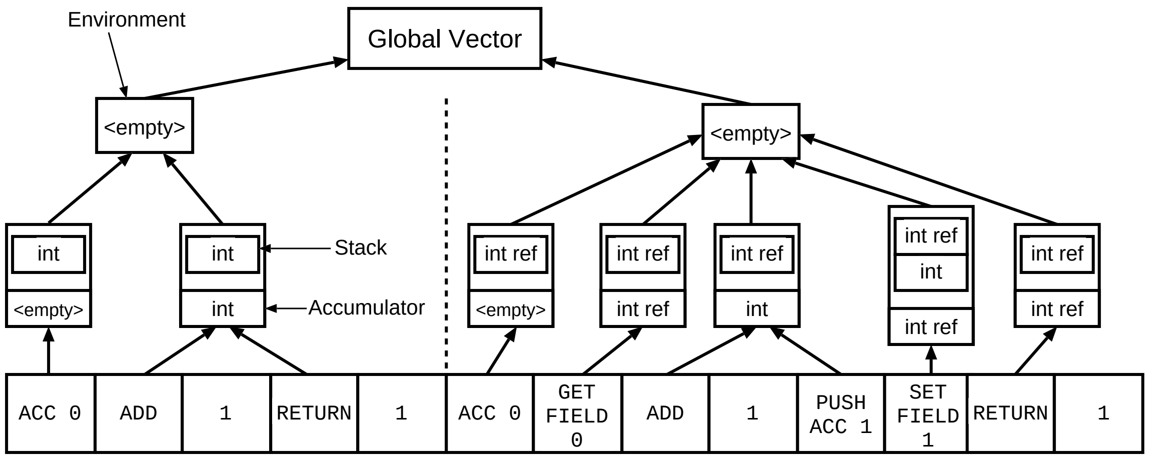


Figure A.2: Verifier state for simple functions

- *universally qualified* types such as α `list` and $\alpha \times \beta \rightarrow \alpha$.
- *unknown* types.

A.3 Verification

The verification process involves ensuring that this set of $(\mathcal{A}, \mathcal{S}, \mathcal{E}, \mathcal{G})$ tuples can be created for all executable instructions in a module, without any inconsistencies. Certain locations in the bytecode – such as the operand locations for instructions spanning multiple bytecode locations – will not be executable and hence do not require a tuple associated with them.

The verifier should maintain a stack of locations to be checked; this will be initialised to

Instruction	Pre state (PC = n)			Post state(s)			
	Acc	Stack	Env	PC	Acc	Stack	Env
`SETFIELD0`	$(\alpha, \ldots)$	$(\alpha, \ldots)$	$\mathcal{E}$	$n+1$	empty	$(\ldots)$	$\mathcal{E}$
`APPLY2`	$\alpha \to \beta \to \gamma$	$(\alpha, \beta, \ldots)$	$\mathcal{E}$	$n+1$	γ	$(\ldots)$	$\mathcal{E}$
`GETGLOBFIELD`	?	$\mathcal{S}$	$\mathcal{E}$	$n+3$	$\mathcal{G}[m][f]$	$\mathcal{S}$	$\mathcal{E}$
`ACC2`	?	$(\alpha, \beta, \gamma, \ldots)$	$\mathcal{E}$	$n+1$	γ	$(\alpha, \beta, \gamma, \ldots)$	$\mathcal{E}$
`MAKEBLOCK3`	α	$(\beta, \gamma, \ldots)$	$\mathcal{E}$	$n+1$	(α, β, γ)	$(\ldots)$	$\mathcal{E}$
`BRANCHIF`	`int`	$\mathcal{S}$	$\mathcal{E}$	$n+2$	`int`	$\mathcal{S}$	$\mathcal{E}$
"	"	"	"	$n+o$	`int`	$\mathcal{S}$	$\mathcal{E}$
`ADDINT`	`int`	(`int`,...)	$\mathcal{E}$	$n+1$	`int`	$(\ldots)$	$\mathcal{E}$
`ENVACC1`	?	$\mathcal{S}$	$(\alpha, \ldots)$	$n+1$	α	$\mathcal{S}$	$(\alpha, \ldots)$

Figure A.3: Verifier logic for a subset of the OCaml virtual machine

contain the first location of the module's initialisation code. To verify a particular location,the possible set of (PC $\times$ state) pairs must be calculated. For each pair in this set the tuple at the new PC must be unified with the new state, possibly specialising any unknown or qualified types in the new PC's tuple, and the new PC must be pushed on to the location stack if it has not already been processed, or if the unification process has resulted in a changed state for the new PC. In the case of an instruction that creates a new closure, the PC of the closure must also be pushed on to the location stack, along with any information known about the types of values in its environment and parameters. If the unification stage fails, then the bytecode does not respect the virtual machine's type system and should be rejected.

The transformation from (PC, state) to (PC$'$, state$'$) is specific to the instruction at the location being verified. Figure A.3 gives this transformation for a representative subset of the OCaml virtual machine instruction set:

`SETFIELD0` updates the value of the first field in a record; the destination record is found in the accumulator, and the new value is on the top of the stack.

`APPLY2` calls a two argument closure; the closure is found in the accumulator and the two arguments are on the top of the stack.

`GETGLOBFIELD` obtains a value from the global vector. The module m and the field f are taken from the two bytecode locations following the instruction.

`ACC2` loads the value at stack position 2 into the accumulator.

`MAKEBLOCK3` allocates a new heap block that is three words long. The block is initialised from the values in the accumulator and the top two stack locations.

`BRANCHIF` branches if the integer value in the accumulator is non-zero. The destination offset o is taken from the bytecode location following the instruction.

`ADDINT` adds the value on the top of the stack to the accumulator.

`ENVACC1` loads the value at environment position 1 into the accumulator. In the OCaml heap, a closure and its environment are the same object – the first value in the object is the bytecode location of the function referred to by the closure, and the subsequent values constitute the environment of the closure. Therefore environment position 1 is actually the first value in the environment proper.

State transformations for other OCaml VM instructions may be constructed in order to complete the design for the verifier.

Bibliography

[ABone] *Active Network Backbone.* `http://www.isi.edu/abone`. (p 145)

[Alexander97a] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. *Active Network Encapsulation Protocol (ANEP).* `http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt`, August 1997. (pp 22, 145)

[Alexander97b] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. *Active Bridging.* In SIGCOMM [SIG97]. (p 25)

[Alexander98a] D. Scott Alexander. *ALIEN: A Generalized Computing Model of Active Networks.* Ph.D. Dissertation, University of Pennsylvania, September 1998. (pp 24, 29, 30, 55, 94, 99)

[Alexander98b] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. *The SwitchWare Active Network Architecture.* IEEE Network Magazine Special Issue on Active and Controllable Networks, 12(3):29–36, May 1998. (p 30)

[Alexander98c] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. *A Secure Active Network Environment Architecture.* IEEE Network Magazine Special Issue on Active and Controllable Networks, 12(3):37–45, May/June 1998. (pp 30, 32, 60)

[Amir98] Elan Amir, Steven McCanne, and Randy H. Katz. *An Active Service Framework and its Application to real-time Multimedia Transcoding.* In SIGCOMM [SIG98]. (pp 21, 23, 26)

[ATMF96] *Private Network-Network Interface Specification Version 1.0 (PNNI 1.0)*, March 1996. ATM Forum. (p 23)

[Back99] Godmar Back and Wilson Hsieh. *Drawing the Red Line in Java.* In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII), March 1999. (p 56)

[Back00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.* Technical Report UUCS-00-010, Department of Computer Science, University of Utah, April 2000. (pp 37, 73)

[Banga99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. *Resource containers: A new facility for resource management in server systems.* In OSDI [OSD99], pages 45–58. (pp 35, 49, 51)

[Barham96] Paul Barham. *Devices in a Multi-Service Operating System.* PhD Thesis 403, University of Cambridge Computer Laboratory, October 1996. (pp 48, 50, 51, 52, 78)

[Barham97] Paul Barham. *A Fresh Approach to File System Quality of Service.* In Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97), 1997. (p 162)

[Barham98] Paul Barham, Simon Crosby, Tim Granger, Neil Stratford, Fergal Toomey, and Muriel Huggard. *Measurement Based Admission Control and Resource Allocation for Multimedia Applications.* In Proceedings of the 1998 IEEE Conference on Multimedia Computing and Networking (MMCN '98), January 1998. (p 163)

[Bernadat98] Philippe Bernadat, Dan Lambright, and Franco Travostino. *Towards a Resource-safe Java.* In Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications (PLRTIA), December 1998. (pp 37, 73, 74)

[Bershad90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. *Lightweight Remote Procedure Call.* ACM Transactions on Computer Systems, 8(1):37–55, February 1990. (p 77)

[Bershad95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. *Extensibility, Safety and Performance in the SPIN Operating System.* In SOSP [SOS95], pages 267–284. (p 33)

[Bhattacharjee97] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. *Active Networking and the End-To-End Argument.* In Proceedings of the IEEE International Conference on Network Protocols, October 1997. (pp 22, 182)

[Bhattacharjee98] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. *Congestion Control and Caching in CANES.* In Proceedings of the IEEE International Conference on Caching (ICC '99), 1998. (p 25)

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1):39–59, February 1984. (pp 76, 77)

[Birrell93] Andrew Birrell, Greg Nelson, Susan Owicki, and Ted Wobber. *Network Objects.* In Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93), volume 27(5) of *ACM Operating Systems Review*, pages 217–230, December 1993. (pp 77, 79, 85, 87)

[Black94] Richard Black. *Explicit Network Scheduling.* PhD Thesis 361, University of Cambridge Computer Laboratory, December 1994. (pp 44, 51, 78)

[Black97] 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 '97), 1997. (pp 36, 47, 109)

[Blake98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *RFC 2475: An Architecture for Differentiated Services*, December 1998. (pp 20, 39)

[Blaze99] M. Blaze, J. Feigenbaum, and A. D. Keromytis. *KeyNote: Trust Management for Public-Key Infrastructures.* Lecture Notes in Computer Science, 1550:59–63, 1999. (pp 33, 60)

[Borenstein94] Nathaniel S. Borenstein. *E-mail with a mind of its own: The Safe-Tcl language for enabled mail.* IFIP Transactions C. Comm Syst, 25:389–402, 1994. (p 30)

[Bos99] Herbert Bos. *Elastic Network Control.* Ph.D. Dissertation, University of Cambridge Computer Laboratory, August 1999. (pp 25, 26, 30, 60)

[Braden94] R. Braden, D. Clark, and S. Shenker. *RFC 1633: Integrated Services in the Internet Architecture: an Overview*, June 1994. (p 20)

[Braden99] B. Braden, A. Cerpa, T. Faber, B. Lindell, G. Phillips, and J. Kann. *An Active Execution Environment for Network Control Protocols.* October 1999. (p 24)

[Bredin98] Jonathan Bredin, David Kotz, and Daniela Rus. *Market-based Resource Control for Mobile Agents.* In Proceedings of the Second International Conference on Autonomous Agents, pages 197–204. ACM Press, May 1998. (pp 36, 89)

[Breslau98] Lee Breslau and Scott Shenker. *Best-Effort versus Reservations: A Simple Comparative Analysis.* In SIGCOMM [SIG98]. (pp 39, 41)

[Bruno99] John Bruno, Jose Brustoloni, Eran Gabber, Bann Ozden, and Abraham Silberschatz. *Retrofitting Quality of Service into a Time-Sharing Operating System.* In Proceedings of the 1999 USENIX Annual Technical Conference, pages 15–26, June 1999. (pp 35, 49, 162)

[Brustoloni00] Jose Brustoloni, Eran Gabber, Abraham Silberschatz, and Amit Singh. *Signaled Receiver Processing.* In Proceedings of the 2000 USENIX Annual Technical Conference, June 2000. (pp 34, 51)

[Bugnion97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. *Disco: Running Commodity Operating Systems on Scalable Multiprocessors.* In SOSP [SOS97], pages 143–156. (p 61)

[Callon97] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan. *A Framework for Multiprotocol Label Switching*, July 1997. Internet Draft. (p 20)

[Calvert98] *K Calvert, ed. Architectural Framework for Active Networks. AN Architecture Working Group*, June 1998. Draft. (pp 27, 162)

[Campbell99] Andrew T. Campbell, Herman G. De Meer, Michael E. Kouvanis, Kazuho Miki, John B. Vicente, and Daniel Villela. *A Survey of Programmable Networks.* Computer Communication Review, 29(2):7–23, April 1999. (p 21)

[Cardelli89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report (revised).* Research Report 52, Compaq (formerly DEC) Systems Research Centre, September 1989. (p 33)

[Cardelli94] Luca Cardelli. *Obliq: A Language with Distributed Scope.* Technical Report, Compaq (formerly DEC) Systems Research Centre, 1994. (p 31)

[Cardoe99] R. Cardoe, J. Finney, A.C. Scott, and W.D. Shepherd. *LARA: A Prototype System for Supporting High Performance Active Networking.* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 117–131. Springer-Verlag, July 1999. (p 27)

[Case90] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. *RFC 1157: Simple Network Management Protocol (SNMP)*, May 1990. (pp 25, 30)

[CERT96] *TCP SYN Flooding and IP Spoofing Attacks.* CERT Advisory, September 1996. `http://www.cert.org/advisories/CA-96.21.tcp_syn_flooding.html`. (p 40)

[CERT98] *"Smurf" IP Denial-of-Service Attacks.* CERT Advisory, January 1998. `http://www.cert.org/advisories/CA-98.01.smurf.html`. (p 40)

[CERT99] *Distributed Denial-of-Service Tools.* CERT Advisory, November 1999. `http://www.cert.org/incident-notes/IN-99-07.html`. (p 90)

[Chase93] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. *Sharing and Protection in a Single Address Space Operating System.* Technical Report 93-04-02, revised January 1994, Department of Computer Science and Engineering, University of Washington, April 1993. (p 77)

[Cheshire96] Stewart Cheshire. *Latency and the Quest for Interactivity.* `http://www.bolo.net/cheshire/papers/LatencyQuest.ps`, November 1996. (p 13)

[Cohen99] Ariel Cohen and Sampath Rangarajan. *A Programming Interface for Supporting IP Traffic Processing.* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of

Lecture Notes in Computer Science, pages 132–143. Springer-Verlag, July 1999. (p 27)

[Cramer97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seherger, Robert Wilson, and Mario Wolczko. *Compiling Java Just in Time: Using runtime compilation to improve Java program performance.* IEEE Micro, 17(3), May 1997. (pp 62, 99)

[Creasy81] R. J. Creasy. *The origin of the VM/370 time-sharing system.* IBM Journal of Research and Development, 25(5):483–490, September 1981. (p 61)

[Crosby95] Simon Crosby. *Performance Management in ATM Networks.* PhD Thesis 393, University of Cambridge Computer Laboratory, April 1995. (p 23)

[Crovella95] Mark Crovella and Azer Bestavros. *Explaining World Wide Web Traffic Self-Similarity.* Technical Report 95-015, Boston University, August 1995. (p 39)

[Cugola97] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. *Analyzing Mobile Code Languages.* Lecture Notes in Computer Science, 1222:93–110, April 1997. (p 29)

[Czajkowski98] Grzegorz Czajkowski and Thorsten von Eicken. *JRes: A Resource Accounting Interface for Java.* In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Langauges and Applications (OOPSLA '98), November 1998. (pp 37, 72)

[DARPA] *Active Networking Working Group discussion list archives.* `http://www.ittc.ukans.edu/Projects/ActiveNets/`. (pp 176, 179, 181)

[Decasper98] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. *Router Plugins: A Software Architecture for Next Generation Routers.* In SIGCOMM [SIG98], pages 229–240. (p 27)

[Deering89] S. E. Deering. *RFC 1112: Host extensions for IP multicasting*, August 1989. (pp 21, 153)

[Deering95] Steve Deering, Scott Shenker, Dave Clark, Christian Huitema, and Domenico Ferrari. *Reservations or No Reservations?* `ftp://ftp.parc.xerox.com/pub/net-research/infocom95.html`, April 1995. Infocom '95 panel discussion (slides). (p 39)

[Demers89] Alan Demers, Srinivasan Keshav, and Scott Shenker. *Analysis and Simulation of a Fair Queueing Algorithm.* In Proceedings of ACM SIGCOMM Conference, volume 19(4) of *Computer Communication Review*, pages 1–12, September 1989. (pp 47, 64)

[Doligez93] Damien Doligez and Xavier Leroy. *A concurrent generational garbage collector for a multithreaded implementation of ML.* In Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL '93), 1993. (p 104)

[Druschel96] Peter Druschel and Gaurav Banga. *Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems.* In OSDI [OSD96], pages 261–275. (pp 34, 49, 67)

[Engler95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. *Exokernel: an Operating System Architecture for Application-Level Resource Management.* In SOSP [SOS95]. (pp 34, 50)

[Engler96] Dawson R. Engler and M. Frans Kaashoek. *DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation.* In SIGCOMM [SIG96], pages 53–59. (p 34)

[Floyd97] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. *A reliable multicast framework for light-weight sessions and application level framing.* IEEE/ACM Transactions on Networking, 5(6):784–803, December 1997. (pp 153, 154, 155)

[Ford94] Bryan Ford and Jay Lepreau. *Evolving Mach 3.0 to A Migrating Thread Model.* In Proceedings of the Winter 1994 USENIX Conference, pages 97–114. USENIX, January 1994. (p 77)

[Fraser93] A. G. Fraser. *Early Experiments with Asynchronous Time Division Networks.* IEEE Network Magazine, 7(1):12–26, January 1993. (p 20)

[Fry98] M. Fry and A. Ghosh. *Application Layer Active Networking.* In Proceedings of the Fourth International Workshop on High Performance Protocol Architectures (HIPPARCH '98), June 1998. (pp 23, 26)

[Glassman95] Steve Glassman, Mark Manasse, Martin Abadi, Paul Gauthier, and Patrick Sobalvarro. *The Millicent protocol for inexpensive electronic commerce.* World Wide Web Journal, 1(1), December 1995. (p 89)

[Goldberg74] Robert P. Goldberg. *Survey of Virtual Machine Research.* IEEE Computer Magazine, 7(6):34–45, June 1974. (p 61)

[Gosling95a] J. Gosling. *Java intermediate bytecodes.* ACM SIGPLAN Notices, 30(3):111–118, March 1995. (pp 61, 100)

[Gosling95b] J. Gosling and H. McGilton. *The Java Language Environment. A White Paper.* Sun Microsystems, 1995. (pp 14, 24, 29)

[Gray93] J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufmann Publishers, 1993. (p 82)

[Hadzic98] Ilija Hadzic and Jonathan M. Smith. *On-the-fly Programmable Hardware for Networks.* In Proceedings of the IEEE Global Telecommunications Conference, November 1998. (pp 21, 23, 25)

[Halls97] David Halls. *Applying Mobile Code to Distributed Systems.* Ph.D. Dissertation, University of Cambridge Computer Laboratory, June 1997. (p 30)

[Hand99] Steven Hand. *Self-Paging in the Nemesis Operating System.* In OSDI [OSD99]. (pp 36, 162)

[Harris98] Timothy Harris. *Controlling Run-time Compilation.* In Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications, pages 75–84, December 1998. (pp 62, 99)

[Harris99] Timothy Harris. *An Extensible Virtual Machine Architecture.* In Proceedings of the OOPSLA '99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, November 1999. (p 65)

[Härtig97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. *The Performance of μ-Kernel-based Systems.* In SOSP [SOS97], pages 66–77. (pp 49, 132)

[Hartman98] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. *Joust: A Platform for Liquid Software.* IEEE Network Magazine, 12(4), July 1998. (p 28)

[Hawblitzel98] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java.* In Proceedings of the 1998 USENIX Annual Technical Conference, June 1998. (pp 37, 74, 77, 85, 131, 132)

[Hayton96] Richard Hayton and Ken Moody. *An Open Architecture for Secure Interworking Services.* In Proceedings of the Seventh ACM SIGOPS European Workshop, September 1996. (pp 33, 60)

[Hicks98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. *PLAN: A Packet Language for Active Networks.* In Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages, pages 86–93. ACM, 1998. (pp 24, 94, 146)

[Hicks99a] Michael Hicks. *Email to ActiveNets_Wire (AN Working Group discussion list [DARPA]),* 5th October 1999. Subject: *Re: Questions for the community.* (p 30)

[Hicks99b] Michael Hicks and Angelos D. Keromytis. *A Secure PLAN.* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, July 1999. (pp 32, 60, 64)

[Hicks99c] Michael Hicks, Jonathan Moore, D. Scott Alexander, Carl Gunter, and Scott Nettles. *PLANet: An Active Internetwork.* In Proceedings of IEEE Infocom '99, March 1999. (pp 24, 30, 46, 59, 108, 119, 146)

[Hjalmtysson99] Gisli Hjalmtysson and K.K. Ramakrishnan. *UNITE - An Architecture for Lightweight Signalling in ATM Networks.* In Proceedings of IEEE Infocom '98, pages 832–840, March 1999. (p 23)

[Hoare73] C. A. R. Hoare. *Monitors: an Operating System Structuring Concept.* Technical Report CS-TR-73-401, Stanford University, Department of Computer Science, November 1973. (p 76)

[Hornof99] Luke Hornof and Trevor Jim. *Certifying Compilation and Run-time Code Generation.* In Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99), January 1999. (p 32)

[Ingram99] David Ingram. *Soft Real Time Scheduling for General Purpose Client-Server Systems.* In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII), March 1999. (pp 34, 49, 78)

[ITU-T94] ITU-T. *Draft Recommendation Q.2931, Broadband Integrated Service Digital Network (B-ISDN) Digital Subscriber Signalling Systems No. 2, User-Network Interface layer 3 specification for basic call/connection control.* ITU publication, November 1994. (p 23)

[Jabbari91] Bijan Jabbari. *Common Channel Signalling System Number 7 for ISDN and Intelligent Networks.* Proceedings of the IEEE, 79(2):155–169, 1991. (p 23)

[Jacobsen80] T. Jacobsen and P. Thisted. *CCITT Recommendation X.25 as Part of the Reference Model of Open Systems Interconnection.* Computer Communication Review, 10(1–2):56–129, January 1980. (p 20)

[Johansen95] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. *Operating System Support for Mobile Agents.* In Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HOTOS-V), pages 42–45, May 1995. (p 30)

[Jones97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities.* In SOSP [SOS97], pages 198–211. (p 81)

[Kaashoek97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. *Application Performance and Flexibility on Exokernel Systems.* In SOSP [SOS97], pages 52–65. (p 34)

[Karjoth98] G. Karjoth, N. Asokan, and C. Gülcü. *Protecting the Computation Results of Free-Roaming Agents.* In K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 195–207. Springer-Verlag, 1998. (p 60)

[Kotz97] D. Kotz et al. *Agent TCL: Targeting the Needs of Mobile Computers.* IEEE Internet Computing, 1(4):50–59, July 1997. (p 30)

[Kulkarni98] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, A. Gopinath, S. Sheth, F. Wahab, H.Pindi, and A. Nagarajan. *Implementation of a Prototype*

Active Network. In Proceeding of the 1st IEEE Conference on Open Architectures and Network Programming (OPENARCH '98), April 1998. (pp 24, 30)

[Lal99] Manoj Lal and Raju Pandey. *CPU Resource Control for Mobile Programs.* In Proceedings of the First International Symposium on Agent Systems and Applications (ASA '99)/Third International Symposium on Mobile Agents (MA '99), October 1999. (p 36)

[Lampson80] Butler W. Lampson and David D. Redell. *Experience with Processes and Monitors in Mesa.* Communications of the ACM, 23(2):105–117, February 1980. (pp 80, 135)

[Lauer79] H. C. Lauer and R. M. Needham. *On the Duality of Operating System Structures.* In Proceedings of the 2nd International Symposium on Operating System Principles, IRIA, volume 13(2), page 3, April 1979. (p 76)

[Lazar96] A. A. Lazar, K. S. Lim, and F. Marconcini. *Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture.* IEEE Journal on Selected Areas in Communications, 14(7):1214–1227, September 1996. (p 23)

[Lee99] David C. Lee, Mark T. Jones, Scott F. Midkiff, and Peter M. Athanas. *Towards Active Hardware.* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 180–187. Springer-Verlag, July 1999. (pp 23, 25)

[Legedza98] Ulana Legedza and John Guttag. *Using Network Level Support to Improve Cache Routing.* In Proceedings of the 3rd International WWW Caching Workshop, June 1998. (p 21)

[Lehman98] Li-wei Lehman, Stephen J. Garland, and David L. Tennenhouse. *Active Reliable Multicast.* In Proceedings of IEEE Infocom '98, March 1998. (pp 21, 154, 155)

[Leland93] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. *On the self-similar nature of Ethernet traffic.* In Proceedings of ACM SIGCOMM Conference, volume 23(4) of *Computer Communication Review*, pages 183–193, September 1993. (p 39)

[Lepreau99] Jay Lepreau et al. *Janos: A Java-oriented Active Network Operating System*, December 1999. `http://www.cs.utah.edu/flux/janos/`. (p 28)

[Leroy97] Xavier Leroy. *Objective Caml.* INRIA, 1997. `http://caml.inria.fr/ocaml/`. (pp 24, 30, 63, 92, 165)

[Leslie96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. *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. (pp 35, 48, 50, 92)

[Liu73] C. Liu and J. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment.* Journal of the ACM, 20(1):46–61, February 1973. (pp 44, 50, 64, 100)

[Lucent96] Lucent Technologies Inc. *Inferno: la Commedia Interattiva*, 1996. `http://inferno.bell-labs.com/inferno/infernosum.html`. (p 30)

[Mankin98] A. Mankin, A. Romanow, S. Bradner, and V. Paxson. *RFC 2357: IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols*, June 1998. (p 153)

[Marshall99] Ian Marshall, Mike Fry, Luis Velasco, and Atanu Ghosh. *Active Information Networks and XML.* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 60–72. Springer-Verlag, July 1999. (p 26)

[Menage98a] Paul Menage. *A byte code for flexible dynamic generation of marshalling routines.* Submitted to the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, March 1998. (pp 51, 115)

[Menage98b] Paul Menage. *Email to ActiveNets_NodeOS (AN Working Group discussion list [DARPA])*, 9th August 1998. Subject: *Re: draft spec.* (p 64)

[Menage99] Paul Menage. *RCANE: A Resource Controlled Framework for Active Network Services.* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, July 1999. (pp 30, 54)

[Merugu00] Shashidhar Merugu, Samrat Bhattacharjee, Ellen Zegura, and Ken Calvert. *Bowman: A Node OS for Active Networks.* In Proceedings of IEEE Infocom 2000, March 2000. (pp 28, 29, 37, 46)

[Milner97] Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. *The Standard ML Programming Language (Revised).* MIT Press, 1997. (pp 30, 72, 94)

[Moore99a] Jonathan Moore. *Email to ActiveNets_Wire (AN Working Group discussion list [DARPA])*, 30th September 1999. Subject: *Re: Questions for the community.* (p 30)

[Moore99b] Jonathan T. Moore, Michael Hicks, and Scott M. Nettles. *Chunks in PLAN: Language Support for Programs as Packets.* Technical Report, Department of Computer and Information Science, University of Pennsylvania, April 1999. (pp 30, 146)

[Morris99] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. *The Click Modular Router.* In SOSP [SOS99], pages 217–231. (pp 27, 35)

[Morrisett98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. *From System F to Typed Assembly Language.* In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98), San Diego, California, pages 85–97. ACM, January 1998. (p 32)

[Mosberger96a] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. *Analysis of Techniques to Improve Protocol Processing Latency.* In SIGCOMM [SIG96], pages 73–84. (p 35)

[Mosberger96b] David Mosberger and Larry L. Peterson. *Making Paths Explicit in the Scout Operating System.* In OSDI [OSD96], pages 153–167. (p 35)

[Murphy97] D. Murphy. *Building an Active Node on the Internet.* Technical Report MIT/LCS/TR-723, Massachusetts Institute of Technology, May 1997. (p 21)

[Necula97] G. C. Necula. *Proof-carrying code.* In Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL '97), pages 106–119, January 1997. (pp 32, 41, 65)

[Necula98] G. C. Necula and P. Lee. *Safe, Untrusted Agents using Proof-Carrying Code.* In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998. (pp 37, 41, 61)

[Newman96] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. Ching Liaw, T. Lyon, and G. Minshall. *RFC 1987: Ipsilon's General Switch Management Protocol Specification Version 1.1.* August 1996. (p 23)

[Nieh94] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. *SVR4 UNIX Scheduler Unacceptable for Multimedia Applications.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '94), volume 846 of *Lecture Notes in Computer Science*, pages 41–53. Springer-Verlag, 1994. (p 43)

[Nilsen98] Kelvin Nilsen. *Adding real-time capabilities to Java.* Communications of the ACM, 41(6):49–56, June 1998. (p 81)

[Nipkow99] Tobias Nipkow. *Towards Verified Bytecode Verifiers.* `http://www4.informatik.tu-muenchen.de/~nipkow/pubs/bcv.ps.gz`, 1999. (pp 100, 112)

[NIST88] National Institute of Standards and Technology (U. S.). *POSIX: portable operating system interface for computer environments*, September 1988. Shipping list no.: 88-752-P. (p 26)

[NIST95] National Institute of Standards and Technology, US Department of Commerce. *FIPS 180-1, Secure Hash Standard*, April 1995. (p 24)

[Nygren99] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek. *PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems.* In Proceeding of the 2nd IEEE Conference on Open Architectures and Network Programming (OPENARCH '99), pages 78–89, March 1999. (pp 24, 27, 29, 30)

[OMG98] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998. (p 85)

[OMG] Object Management Group. *CORBA Services Specification*, December 98. `http://www.omg.org/corba/sectran1.html`. (p 87)

[ORL97] Olivetti-Oracle Research Laboratory. *OmniORB*, 1997. `http://www.orl.co.uk/omniORB/omniORB.html`. (p 87)

[OSD96] *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96).* USENIX, October 1996. (pp 175, 180, 183)

[OSD99] *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '98).* USENIX, February 1999. (pp 171, 176)

[Ousterhout90] John K. Ousterhout. *Tcl: An Embeddable Command Language.* In Proceedings of the Winter 1990 USENIX Conference, pages 133–146. USENIX, January 1990. (pp 26, 30)

[Paxson94] V. Paxson and S. Floyd. *Wide-Area Traffic: The Failure of Poisson Modeling.* In Proceedings of ACM SIGCOMM Conference, volume 24(4) of *Computer Communication Review*, pages 257–268, September 1994. (p 39)

[Peterson00a] L Peterson, ed. *NodeOS Interface Specification. AN Node OS Working Group*, January 2000. Draft. (pp 27, 37, 57, 64, 145)

[Peterson00b] Larry Peterson. *Email to ActiveNets_Wire (AN Working Group discussion list [DARPA])*, 10th March 2000. Subject: *Re: NodeOS questions.* (p 145)

[Postel81] J. Postel. *RFC 791: Internet Protocol*, September 1981. (pp 19, 21)

[Pratt97] Ian Pratt. *The User-Safe Device I/O Architecture.* Ph.D. Dissertation, University of Cambridge Computer Laboratory, August 1997. (pp 52, 68)

[Pratt00] Ian Pratt and Keir Fraser. *The* ArseNIC *User-Safe Ethernet Card.* March 2000. (p 68)

[Reed99] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. *Xenoservers: Accountable Execution of Untrusted Programs.* In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII), March 1999. (p 26)

[Ritchie83] Dennis M. Ritchie and Ken Thompson. *The UNIX Time-Sharing System.* Communications of the ACM, 26(1):84–89, January 1983. (p 43)

[Rivest92] R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*, April 1992. (pp 24, 150)

[Romanenko99] Sergei Romanenko and Peter Sestoft. *Moscow ML*, 1999. `http://www.dina.kvl.dk/~sestoft/mosml.html`. (p 94)

[Rooney97] Sean Rooney. *Connection Closures: Adding Application-defined Behaviour to Network Connections.* Computer Communication Review, April 1997. (p 26)

[Rooney98a] Sean Rooney. *The Structure of Open ATM Control Architectures.* Ph.D. Dissertation, University of Cambridge Computer Laboratory, 1998. (pp 25, 30)

[Rooney98b] Sean Rooney, Jacobus van der Merwe, Simon Crosby, and Ian Leslie. *The Tempest: a Framework for Safe, Resource Assured, Programmable Networks.* IEEE Communications Magazine, 36(10):42–53, October 1998. (p 23)

[Roscoe95] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD Thesis 376, University of Cambridge Computer Laboratory, August 1995. (pp 35, 44, 48, 77, 78, 100)

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. *End-To-End Arguments in System Design.* ACM Transactions on Computer Systems, 2(4):277–288, November 1984. (pp 19, 22)

[Saltzer98] J. H. Saltzer, D. P. Reed, and D. D. Clark. *Comment on 'Active Networking and End-To-End Arguments'.* IEEE Network Magazine, 12(3):69–71, May 1998. (Response to [Bhattacharjee97]). (p 22)

[Sander98] Tomas Sander and Christian F. Tschudin. *Protecting Mobile Agents Against Malicious Hosts.* In Giovanni Vigna, editor, *Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 1998. (p 60)

[Saraswat97] Vijay Saraswat. *Java is Not Type-Safe.* `http://www.research.att.com/~vj/bug.html`, August 1997. (p 112)

[Schwartz99] Beverley Schwartz, Alden Jackson, Timothy Strayer, Wenyi Zhou, Dennis Rockwell, and Craig Partridge. *Smart Packets for Active Networks.* In Proceeding of the 2nd IEEE Conference on Open Architectures and Network Programming (OPENARCH '99), March 1999. (pp 25, 30)

[Searle80] J. Searle. *Minds, Brains, and Programs.* Behavioral & Brain Sciences, 3:417–458, 1980. The classic "Chineese room" argument against AI programs being considered "intelligent". (p 60)

[SecArch98] *Security Architecture for Active Nets*, July 1998. AN Security Working Group Draft. (p 33)

[Seltzer96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions.* In OSDI [OSD96], pages 213–227. (pp 33, 62, 82)

[SIG96] *Proceedings of ACM SIGCOMM Conference*, volume 26(4) of *Computer Communication Review*, August 1996. (pp 175, 180)

[SIG97] *Proceedings of ACM SIGCOMM Conference*, volume 27(4) of *Computer Communication Review*, September 1997. (pp 170, 185)

[SIG98] *Proceedings of ACM SIGCOMM Conference*, volume 28(4) of *Computer Communication Review*, September 1998. (pp 170, 172, 174, 183)

[Smith98] Jonathan M. Smith, D. Scott Alexander, W. S. Marcus, M. Segal, and W. D. Sincoskie. *Towards an Active Internet.* July 1998. (p 25)

[SOS95] *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, volume 29(5) of *ACM Operating Systems Review*, December 1995. (pp 171, 175)

[SOS97] *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, volume 31(5) of *ACM Operating Systems Review*, October 1997. (pp 172, 176, 177)

[SOS99] *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, volume 33(5) of *ACM Operating Systems Review*, December 1999. (pp 179, 185)

[SPEC95] *SPEC CPU95 Benchmark*, 1995. Standard Performance Evaluation Corporation, `http://www.spec.org/osg/cpu95/`. (p 162)

[SRI99] SRI International. *Active Network Daemon*, 1999. `http://www.csl.sri.com/ancors/anetd`. (p 145)

[Srinivasan98] V. Srinivasan, George Varghese, Subash Suri, and Marcel Waldvogel. *Fast Scalable Level Four Switching.* In SIGCOMM [SIG98], pages 191–202. (p 20)

[Steele84] Guy L. Steele. *COMMON LISP: The Language.* Digital Press, 1984. With contributions by Scott E. Fahlman and others. (p 105)

[Stratford99] Neil Stratford and Richard Mortier. *An Economic Approach to Adaptive Resource Management.* In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII), March 1999. (p 89)

[Stratford00] Neil Stratford. *Application-Specific Resource Management.* Ph.D. Dissertation, University of Cambridge Computer Laboratory, 2000. In preparation. (p 163)

[Stroustrup86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986. (p 97)

[SUN88] *RFC 1057: RPC: Remote Procedure Call Protocol specification: Version 2*, June 1988. (p 85)

[Sun98] *Java Remote Method Invocation – Distributed Computing for Java.* Sun Microsystems White Paper, March 1998. `http://java.sun.com/marketing/collateral/javarmi.html`. (p 85)

[Tennenhouse89] David L. Tennenhouse. *Layered Multiplexing Considered Harmful.* In Proceedings of the 1st International Workshop on High-Speed Networks, May 1989. (p 36)

[Tennenhouse97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. *A Survey of Active Network Research.* IEEE Communications Magazine, 35(1):80–86, January 1997. (p 21)

[Tschudin97a] Christian Tschudin. *The Messenger Environment M0 – A Condensed Description.* In Mobile Object Systems: Towards the Programmable Internet, pages 149–156. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222. (p 36)

[Tschudin97b] Christian Tschudin. *Open Resource Allocation for Mobile Code.* In Proceedings of the First Workshop on Mobile Agents (MA '97), April 1997. (pp 36, 89)

[Tschudin99] Christian Tschudin. *An Active Networks Overlay Network (ANON).* In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 156–164. Springer-Verlag, July 1999. (p 25)

[van der Merwe96] Kobus van der Merwe. *Switchlets and Dynamic Virtual ATM Networks.* Submitted to ISINM '97, July 1996. (p 23)

[van der Merwe97] Kobus van der Merwe. *Open Service Support for ATM.* Technical Report, University of Cambridge Computer Laboratory, September 1997. (p 23)

[Wahbe93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. *Efficient Software-Based Fault Isolation.* In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203–216, 1993. (pp 31, 62, 82)

[Wakeman98] Ian Wakeman, Alan Jeffrey, Rory Graves, and Tim Owen. *Designing a Programming Language for Active Networks.* Submitted to HIPPARCH Special Issue on Network and ISDN Systems, 1998. (pp 31, 36)

[Waldspurger94] Carl A. Waldspurger and William E. Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management.* In Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '95), pages 1–11. USENIX, November 1994. (pp 36, 43, 80)

[Waldvogel97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. *Scalable High Speed IP Routing Lookups.* In SIGCOMM [SIG97], pages 25–38. (p 20)

[Wall87] Larry Wall. *Perl – Practical Extraction and Report Language.* Usenet `mod.sources` archives, 1987. (p 105)

[Wetherall96] David J. Wetherall and David L. Tennenhouse. *The ACTIVE IP Option.* In Proceedings of the Seventh ACM SIGOPS European Workshop, September 1996. (p 21)

[Wetherall98] David J. Wetherall, John Guttag, and David L. Tennenhouse. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols.* In Proceeding of the 1st IEEE Conference on Open Architectures and Network Programming (OPENARCH '98), April 1998. (pp 24, 30, 46, 59, 94, 150)

[Wetherall99a] David Wetherall. *Active Network Vision and Reality: Lessons from a Capsule-based System.* In SOSP [SOS99]. (pp 41, 153)

[Wetherall99b] David Wetherall. *If bandwidth is the scarce resource, then scheduling bandwidth is sufficient.* Personal Communication, December 1999. (p 40)

[Wilkes79] Maurice Wilkes and Roger Needham. *The Cambridge CAP Computer and its Operating System.* The Computer Science Library, Operating and Programming Systems Series. Elsevier North Holland Inc., 1979. (p 77)

[Wilson92] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques.* In Proceedings of the International Workshop on Memory Management, September 1992. (pp 70, 74)

[Zhang90] Lixia Zhang. *Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks.* In Proceedings of ACM SIGCOMM Conference, volume 20(4) of *Computer Communication Review*, pages 19–29, September 1990. (p 47)

[Zhang93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. *RSVP: A New Resource ReSerVation Protocol.* Technical Report, Xerox PARC and University of Southern California, March 1993. (pp 20, 39)