

MSci Project Proposal

Network Router Resource Virtualisation

Ross McIlroy

December 17, 2004

1 Introduction

There is considerable interest in applications which transport isochronous data across a network, for example teleconferencing or Voice over IP applications. These applications require a network which provides Quality of Service (QoS) guarantees. While there are various networks which provide some form of QoS, there has been little investigation into the allocation of resources to network flows, and partitioning of resources between flows (so a flow only has access to its own resources and cannot affect the QoS provided to other flows). This project will attempt to produce a network router which can segment its internal resources (such as CPU cycles, memory and network bandwidth) between network flows requesting QoS guarantees, thus providing effective partitioning between different network flows.

2 Research Problem

The Internet was created as a best effort service. Packets are delivered as quickly and reliably as possible, however there is no guarantee as to how long a packet will be in transit, or even whether it will be delivered. There is an increasing interest in the creation of networks which can provide Quality of Service (QoS) guarantees. Such networks would be able to effectively transport isochronous data such as audio, video or teleconference data feeds, since each feed could have guaranteed bounds on latency and throughput.

A number of attempts have been made to tackle Quality of Service issues in computer networks. Various approaches have been investigated, including resource reservation protocols, packet scheduling algorithms, overlay networks and new network architectures. Various networks have been built using these approaches, however one of the problems they face is allocating a fixed proportion of the network router resources to each stream. For example, if a denial of service attack is attempted on one of these streams, the router could be overwhelmed, and not able to provide the quality of service needed by the other streams. Some method is needed for partitioning the router resources among streams, so that traffic on one stream does not affect any of the others.

One option is to have separate routers for each stream. This over-provisioning is obviously very expensive and inflexible, however it is currently the standard commercial method of guaranteeing certain bounds on quality of service to customers. This project will investigate the feasibility of creating a sin-

gle routing system, which is built up of a number of virtual *routlets*. Each QoS flow is assigned its own individual virtual outlet in each physical router along the path it takes in the network. These virtual outlets are scheduled by a virtual machine manager, and have a fixed resource allocation dependent upon the QoS required by the flow. This resource partitioning between outlets should provide a similar level of independence between flows as separate routers but at a much reduced cost. This system should also allow for greater flexibility in the network, with dynamic flow creation and dynamic adjustment of flow QoS requirements.

3 Literature Survey

The aim of this project is to create a router which can allocate a fixed proportion of its resources to each QoS flow it handles. To do this, it is important to first understand how a standard router works. Section 3.1 discusses papers involving both hardware and software routers, which were investigated to understand routing techniques.

There have been a number of efforts which have approached the issue of Quality of Service in the Internet. These efforts tackle different areas required by truly QoS-aware networks, such as flow specification, routing, resource reservation, admission control and packet scheduling. As part of this project proposal, I have investigated efforts in a number of these areas, and will explain in Section 3.2 how this project will build upon these ideas.

This project will make use of virtualisation techniques to partition the individual outlets from each other. There have been a number of virtualisation systems created, each with its own aims and objectives. Section 3.3 discusses the virtualisation techniques and systems which were investigated as part of this proposal.

Some efforts are strongly related to this project as they attempt to provide fully QoS-aware networks with resource partitioning. Section 3.4 discusses the techniques used by these networks, and explains how their approach differs from that taken by this project.

3.1 Router Technology

The first stage in creating a router which provides QoS guarantees is to understand how a standard router operates. There are a number of papers which describe the internal workings of modern IP routers (see [17] or [19] for details). A router's primary function is to forward packets between separate networks. Routers, or collections of routers form the core of an internetwork. Each router must understand the protocols of its connected networks and translate packet formats between these protocols. Packets are forwarded according to the destination address in the network layer (e.g. the IP destination). Each router determines where packets should be forwarded by comparing this destination address with entries in a routing table. However, new services are calling upon routers to do much more than simply forward packets. For example, multicast, voice, video and QoS functions call upon the router to analyse additional information within the packets to determine how they should be handled.

A router consists of three main components: network line cards, which physically connect the router to a variety of different networks; a routing processor, which runs routing protocols and builds the routing tables; and a backplane, which connects the line cards and the routing processor together.

The routing process can be seen more clearly by describing the stages involved in the processing of a

packet by a typical IP router. When a packet arrives from a network, the incoming network interface card first processes the packet according to the data-link layer protocol. The data-link layer header is stripped off; the IP header is examined to determine if it is valid; and the destination address is identified. The router then performs a *longest match* lookup (explained in the following paragraph) in the routing table for this destination address. Once a match has been found, the packet's *time to live* field is decremented, a new header checksum is calculated, and the packet is queued for the appropriate network interface. This interface card then encapsulates the IP packet in an appropriate data-link-level frame, and transmits the packet.

Each router is unable to store a complete routing table for the whole internetwork. However, IP addresses have a form of geographical hierarchy, where nodes on the same network have a similar class of IP address. This allows entries in the routing table to aggregate entries for similar addresses that exit the router on the same interface card. In other words, routers know in detail the addresses of nearby networks, but only know roughly where to send packets for distant networks. The routing table stores address prefix / mask pairs. The address prefix is used to match the entry to the destination address, and the mask is used to specify which digits of the address prefix are important. A longest match lookup tries to find the entry that has the most bits in common with that of the destination address. For example, if the address 120.33.195.2 is matched against 120.0.0.0/8, 120.33.1.4/16, and 120.33.225.0/18 in the routing table, the entry 120.33.225.0/18 will be chosen.

The hardware architecture used in routing systems has evolved over time. Initially, the architecture was based upon that of a typical workstation. Network line cards were connected together with a CPU using a shared bus (e.g. ISA or PCI); and all routing decisions and processing was done by the central CPU. Figure 1 shows the architecture of this type of router. The problem with this approach was the single shared bus. Every packet processed has to traverse this bus twice, once from the incoming network line card to the CPU and once from the CPU to the outgoing line card. Since this is a shared bus, only one packet can traverse it at any time. This restricts the overall number of packets which can be processed by a single router to the bandwidth of the shared bus, which is often lower than that of the sum of the network cards' bandwidths.

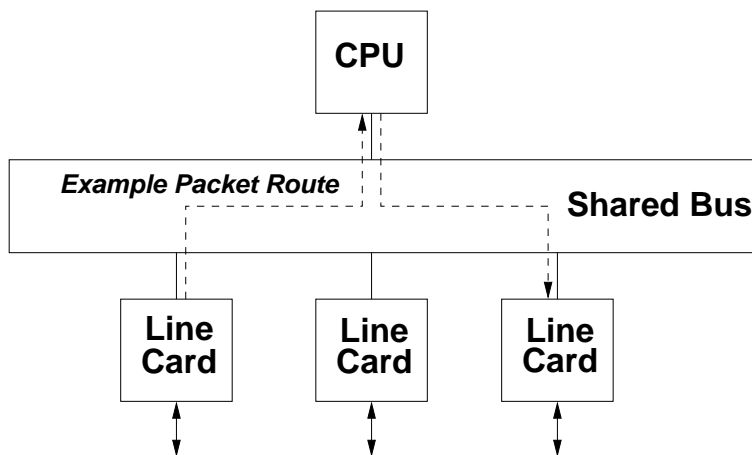


Figure 1: The architecture of a first generation, "Shared Bus" router.

The use of switching fabrics instead of shared busses as a backplane increased the amount of network traffic a router could cope with by allowing packets to traverse the backplane in parallel (Figure 2). The bandwidth of the backplane is no longer shared among the network interface cards. A switching fabric can be implemented in a number of ways; for example, a crossbar switch architecture could be used,

where each line card has a point to point connection to every other line card. Another option is to use shared memory, where each line card can transfer information by writing and reading from the same area of memory.

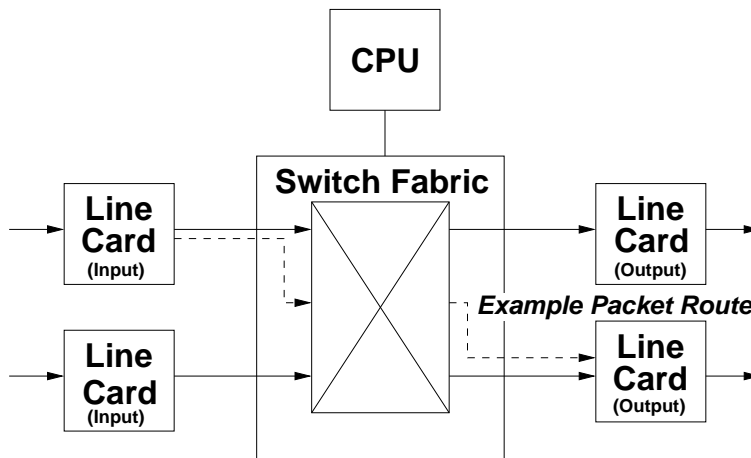


Figure 2: The architecture of a router which uses a switching fabric as its backplane.

The next component which comes under stress from increased network traffic is the shared CPU. A single CPU simply cannot cope with the amount of processing which is needed to support multi gigabit, and therefore multi mega-packet flows. An approach used by many modern routers is to move as much of the packet processing to the line cards as possible. A forwarding engine is placed on each of the network line cards (or in the case of [19], near the NICs). A portion of the network routing table is cached in each forwarding engine, allowing the forwarding engines to forward packets independently. A central processor is still used to control the overall routing, build and distribute the routing tables, and handle packets with unusual options or addresses outside the forwarding engine's cached routing table.

A number of other improvements have been made to router technology, including improved lookup algorithms, advanced buffering techniques and sophisticated scheduling algorithms. The major developments in router hardware, however, have been a movement towards expensive custom hardware, working in parallel, to provide the performance required for multi-gigabit networks.

Software routers run on commodity hardware, and allow a normal PC to perform network routing. Software routers do not have the performance or packet throughput of custom built hardware routers, but they are considerably cheaper and provide more flexibility than hardware routers. They are also an ideal tool for router research, as new protocols, algorithms or scheduling policies can be implemented by simply changing the code, rather than building or modifying hardware components. This project will make use of a software router for these reasons.

The Zebra platform¹ is a popular open source software router. It has initiated a number of spinoff projects, such as Quagga² and ZebOS³, a commercial version of Zebra (all collectively referred to as Zebra hereafter). Zebra runs on a number of UNIX-like operating systems (e.g. Linux, FreeBSD, Solaris etc.) and provides similar routing functionality to commercial routers. The Zebra platform is modular in design. It consists of multiple processes, each of which deals with a different routing protocol. For example, a typical Zebra router will consist of a routing daemon for the open shortest path first (OSPF)

¹<http://www.zebra.org>

²<http://www.quagga.net>

³http://www.ipinfusion.com/products/advanced/products_advanced.html

protocol, and another for the border gateway protocol (BGP). These will work together to create an overall routing table, under the control of a master Zebra daemon. This architecture allows protocols to be added at any time without affecting the whole system.

There have been a number of studies which have compared the features and performance of Zebra to standard commercial hardware routers. Fatoohi et al [12] compared the functionality and usability of the Zebra router with Cisco routers. They find that Zebra provides much of the features found in Cisco routers with a similar user interface. They do not compare the relative performance of the two router types however. Ramanath [22] investigates Zebra's implementation of the BGP and OSPF protocols, but concentrates on the experimental set-up, with little post experimental analysis.

One problem with Zebra is that the open source implementations do not provide MPLS (Multiprotocol Label Switching) support. MPLS routing support is a requirement of this project, so another software router is required. NIST Switch [6] is a research routing platform, specifically created for experimentation in novel approaches to QoS guaranteed routing. It is based on two fundamental QoS concepts, local packet labels which define the characteristics of packets queueing, and traffic shaping measures applied to overall packet routes. NIST Switch supports MPLS, and uses these MPLS labels to define the QoS characteristics and route packets. It also uses RSVP, with Traffic Engineering (TE) extensions, to signal QoS requests and distribute labels. This use of RSVP was also an important requirement in this project as we propose using RSVP to request router resource requests. MPLS and RSVP are explained in more detail in Section 3.2, and their use in this project is described in Section 4.

NIST Switch only runs on the FreeBSD operating system. Lai et al [16] compared Linux, FreeBSD and Solaris running on the x86 architecture. They found that FreeBSD has the highest network performance, making it an ideal operating system on which to base a router. However the difference in network performance cited by Lai et al was not significant enough to make a major difference to the performance of the router created in this project. Linux has also been ported to the para-virtualised, virtual machine monitors required as part of this project, whereas FreeBSD has not. It was therefore decided that a Linux software router was required. Van Heuven et al [14] describes a MPLS capable Linux router, created by merging a number of existing projects, including as a base the NIST Switch router.

A number of modular routers were also investigated as a means of providing additional functionality in a modular fashion. The PromethOS router architecture, described in [15] and based upon the work in [11], provides a modular router framework, which can be dynamically extended using router plugins. The PromethOS framework is based upon the concept of selecting the implementation of the processing components within the router, based upon network flow of the packet currently being processed. When a plugin is loaded, it specifies which packets it will deal with by providing a packet filter. When a packet arrives at a PromethOS router, it traverses the network stack in the usual way. However, as it traverses the network stack, it comes across *gates*, points where the flow of execution can branch off to plugin instances. The packet is compared against the filters specified by the plugin instances, and then sent to the correct plugin. New plugins can be created, and can be inserted at various points along the network stack, using these gates. This allows a variety of different functions to be added to the router, such as new routing algorithms, different queueing technique or enhancements to existing protocols. However it still uses the standard IP network stack. This means that it cannot be used to investigate other network protocols.

The Click Modular Router [18] takes a different approach. Click is not a router itself; instead, it is a software framework which allows the creation of routers. A Click router consists of a number of modules, called elements. These elements are fine-grained components, which perform simple operations such as packet queueing or packet classification. New elements can easily be built by writing a C++ class which corresponds to a certain interface. Elements are *plugged* together using a simple language to

create a certain router configuration. The same elements can be used to create different routers by using this simple language to rearrange the way in which they are organised. These router configurations are compiled down to machine code, which runs within the Linux Kernel to provide a fast, efficient router. The approach taken by the Click router provides a flexible method of creating new routers easily.

3.2 Quality of Service Techniques

There are a number of issues which need to be addressed when creating a QoS-aware network. Firstly the overall routing algorithm needs to take QoS into account, dealing with low quality links, or failures in network nodes. QoS flows also require a mechanism for reservation of routing resources on a network. For a flow to reserve the resources it requires, there needs to be some universal description of the flow's requirements - a flow specification. As network resources are finite, some form of admission control is required to prevent network resources from becoming overloaded. Finally, the packet scheduling algorithms used by the network routers need to be aware of the QoS requirements of each flow, so that bounds on latency and throughput can be guaranteed.

Multiprotocol label switching (MPLS), described in [8] and [25], is a network routing protocol which uses labels to forward packets. This approach attempts to provide the performance and capabilities of layer 2 switching, while providing the scalability and heterogeneity of layer 3 routing. It is therefore often referred to as layer 2.5 routing. These labels also allow an MPLS network to take QoS issues into account while routing packets by, for example, engineering traffic.

In conventional (layer 3) routing, a packet's header is analysed by each router. This information is used to find an element in a routing table, which determines the packet's next hop. In MPLS routing, this analysis is performed only once, at an ingress router. A Label Edge Router (LER), placed at the edge of the MPLS domain, analyses incoming packets, and maps these packets to an unstructured, fixed length, label. Many different headers can map to the same label, as long as these packets all leave the MPLS domain at the same point. Labels, therefore, define *forwarding equivalence classes* (FECs). An FEC defines, not only where a packet should be routed, but also how it should be routed across the MPLS domain.

This labelling process should not interfere with the layer 3 header, or be affected by the layer 2 technology used. The label is therefore added as a *shim* between the layer 2 and the layer 3 headers. This shim is added by the ingress LER as a packet enters the MPLS domain, and removed by the egress LER as it leaves the MPLS domain.

Once a packet has entered the MPLS domain, and has been labelled by the LER, it traverses the domain along a label-switched path (LSP). An LSP is a series of labels along the path from the source to the destination. Packets traverse these LSPs by passing through label switching routers (LSRs). An LSR analyses an incoming packet's label, decides upon the next hop which that packet should take and changes the label to the one used by the next hop for that packet's FEC. The LSR can process high packet volumes, because a label is a fixed length, unstructured value, so a label lookup is fast and simple.

Each LSR (and LER) assigns a different label to the same FEC. This means that globally unique labels are not required for each FEC, as long as they are locally unique. Each LSR needs to know of the labels that neighbouring LSRs use for each FEC. There are a number of ways in which MPLS can distribute this information. The Label Distribution Protocol was designed specifically for this purpose, however it is not commonly used in real networks. Existing routing protocols, such as the Open Shortest Path First (OSPF) or Border Gateway Protocol (BGP), have been enhanced so that label information can be "piggybacked" on the existing routing information. This means that traffic going from a source to a

destination will always follow the same route. Another option is to use RSVP with traffic engineering extensions (RSVP-TE) to explicitly *engineer* routes through the MPLS domain. This allows different routes to be taken by traffic travelling to the same destination depending on its QoS class. For example, a low latency route could be chosen for voice traffic, while best effort traffic, going to the same destination, travels through a more reliable, but slower route.

The Resource ReSerVation Protocol (RSVP) [30] is a protocol which can be used to reserve network resources needed by a packet flow. It reserves resources along simplex, i.e. only single direction, routes. One novel feature of this protocol is that it is receiver-oriented, i.e. the receiver, rather than the sender, initiates the resource reservation, and decides on the resources required. This allows heterogenous data-flows to be accommodated, for example, lower video quality for modem users than for broadband users.

An RSVP session is initiated by a PATH message sent from the sender to the receiver. This PATH message traverses the network, setting up a route between the sender and receiver in a hop-by-hop manner. The PATH message contains information about the type of traffic the sender expects to generate. Routers on the path can modify this data to specify their capabilities. When the receiver application receives the PATH message, it examines this data and generates a message, RESV, which specifies the resources required by the data flow. This RESV message traverses the network in the reverse of the route taken by the PATH message, informing routers along this path of the specified data flow. Each router uses an admission control protocol to decide whether it has sufficient resources to service the flow. If a router cannot service a flow, it will deny the RSVP flow, and inform the sender.

The RSVP protocol is independent of the admission protocol used, so routers can make any decision they wish to decide on the paths it can support. RSVP is also independent of the specification used to describe the flow (the flowspec). As long as the sender, receiver and routers agree on the format, any information can be transmitted in the flowspec to inform routers of the resources required by a flow. RSVP is unaffected by the routing protocol used. PATH messages can be explicitly routed, as in RSVP-TE, and the RESV message follows this route in reverse, thus reserving resources along the explicitly routed path.

RSVP can support dynamic changes in path routes, as well as dynamic changes in the data-flow characteristics of paths. It does this by maintaining soft, rather than hard, state of a reserved path in the network. Data sources periodically send PATH messages to maintain or update the path state in a network. Meanwhile, data receivers periodically send RESV messages to maintain or update the reservation state in the network. If a network router has not received PATH or RESV messages from an RSVP flow for some time, the state of that flow is removed from the router. This makes RSVP robust to network failures.

RSVP was created specifically to support the Quality of Service model called Integrated Services ([3] and [28]). The Integrated Services approach attempts to provide real time QoS guarantees for every flow which requires them. This requires every router to store the state of resources requested by every flow, which could involve a huge amount of data. RSVP addresses this somewhat by having a number of multicast reservation types (e.g. no-filter, fixed-filter and dynamic filter) which allow individual switches to decide how resource reservation requests can be aggregated. However, the state required for an Integrated Service router, especially in the core of the network, is still formidable.

The differentiated services, or Diffserv model ([2] and [31]) takes a different approach. Each packet carries with it information on how it should be treated. For example, a Type of Service (TOS) byte carried in the packet header could specify whether this packet requires low-delay, high-throughput or reliable transport. The problem with this approach is that only simple requirements can be specified (e.g. low latency, rather than, latency less than 10ms) so that each packet does not incur excessive overheads

carrying the QoS requirements. There is also no way to perform admission control, so it is impossible to guarantee the QoS for individual flows. All that can be guaranteed is that certain classes of traffic will be treated better than other classes.

MPLS is in the middle ground between the two extremes of Integrated Services and Diffserv. It provides the capability to set up flows of traffic with certain requirements, much like Integrated Services, however Forwarding Equivalency Classes can be used to aggregate common flows together. Also, the use of unstructured labels greatly simplifies the filtering of traffic into flows, compared with the filtering on source and destination address (as well as other possible IP header fields) used by Integrated Service.

This is a simplified view of Internet QoS techniques, with specific emphasis on those aspects of the technologies which will be used in this project. A more detailed survey is provided by Xiao et al [29].

3.3 Virtualisation Techniques

This project will use virtualisation techniques to partition resources between virtual routelets. Virtualisation was developed by IBM in the mid 1960's specifically for mainframe computers. Currently, there are a number of virtualisation systems available, each of which addresses different issues. Hardware constraints require that the router created by this project must run on standard, commodity x86 hardware. The x86 architecture is an inherently difficult platform to virtualise due to a lack of virtualisation-friendly features in the processor. This proposal, therefore, concentrates on x86 virtualisation techniques as opposed to more general virtualisation techniques.

Virtualisation software allows a single machine to run multiple independent *guest* operating systems concurrently, in separate virtual machines. These virtual machines provide the illusion of an isolated physical machine for each of the guest operating systems. A Virtual Machine Monitor (VMM) takes complete control of the physical machine's hardware, and controls the virtual machines' access to it.

VMware [24], and other full-virtualisation systems, attempt to fully emulate the appearance of the underlying hardware architecture in each of the virtual machines. This allows unmodified operating systems to run within a virtual machine, but presents some problems. For example, operating systems typically expect unfettered, privileged access to the underlying hardware, as it is usually their job to control this hardware. However, guest OSs within a virtualised system cannot have privileged mode access, otherwise they could interfere with other guest OS instances by, for example, modifying their virtual memory page tables. The VMM, on the other hand, controls the underlying physical hardware in a virtualised machine, and therefore has privileged mode CPU access. The guest OSs must be given the appearance of having access to privileged CPU instructions, without being able to harm other OSs.

Fully virtualised systems, such as VMware, typically approach this problem by allowing guest OSs to call these privileged instructions, but by having them handled by the VMM when they are called. The problem is that the x86 fails silently when a privileged instruction is called within a non-privileged environment, rather than providing a trap to a handling routine. This means that VMware has to dynamically rewrite portions of the guest OS kernel at runtime, inserting calls to the VMM where privileged instructions are needed. VMware also keeps shadow copies of each guest OS's virtual page tables, to ensure that guest OS instances do not overwrite each other's memory. However this requires a trap to the VMM for every update of the page table. These requirements mean that an operating system running within a fully virtualised system will have severely reduced performance, compared to the same operating system running on the same physical hardware without the virtualisation layer.

A number of projects have attempted to reduce this performance overhead, at the cost of full physical

architecture emulation. Para-virtualisation is an approach where the virtual machines do not attempt to fully emulate the underlying hardware architecture of the machine being virtualised. Instead, an idealised interface is presented to the guest operating systems, which is similar, but not identical to that of the underlying architecture. This interface does not include the privileged instructions which cause problems in fully virtualised systems, replacing them with calls which transfer control to the VMM (sometimes known as Hypercalls). This change in interface means that existing operating systems need to be ported to this new interface before they will run as guest operating systems. However, the similarity between this interface and the underlying architecture, means that the application binary interface (ABI) remains unchanged, so application programs can run unmodified on the guest OSs. The idealised interface substantially reduces the performance overhead of virtualisation, and also provides the guest OSs with more information. For example, information on both real and virtual time allows better handling of time sensitive operations, such as TCP timeouts.

The Denali Isolation Kernel ([26] and [27]) uses these para-virtualisation techniques to provide an architecture which allows a large number of untrusted Internet services to be run on a single machine. The architecture provides *isolation kernels* for each untrusted Internet service. These isolation kernels are small guest operating systems which run in separate virtual machines, providing partitioning of resources and achieving safe isolation between untrusted application code. The use of simple guest operating systems, and the simplified virtual machine interface, are designed to support Denali's aim of scaling to more than 10,000 virtual machines in one, commodity hardware, machine.

The problem with using Denali, however, is that the interface has been oversimplified to support the scaling needed to run tens of thousands of virtual machines. There have been a number of omissions made in the application binary interface, compared with the x86 architecture. For example, memory segmentation is not supported. These features are expected by standard x86 operating systems and applications. It would therefore be very difficult to port standard operating systems (such as Linux or FreeBSD) to Denali, and even if they were ported, many applications would also have to be rewritten to run on Denali. The Denali project has therefore provided a simple guest OS called Ilwaco which runs under a Denali virtual machine. Ilwaco is simply implemented as a library, much like a Exokernel libOS, with applications linking against the OS. There is no hardware protection boundary in Ilwaco, so essentially each virtual machine hosts a single-user, single-application, unprotected operating system. Since Ilwaco was created specifically for Denali, and is so simple, there is no routing software available for it. Therefore, if Denali is used as the visualisation system in this project, a software router would need to be written from scratch.

Xen [1] is an x86 virtual machine monitor that uses para-virtualisation techniques to reduce the virtualisation overhead. Although it uses an idealised interface like Denali, this interface was specifically created to allow straightforward porting of standard operating systems, and provide an unmodified application binary interface. A number of common operating systems, such as Linux and NetBSD, have been ported to the Xen platform, and because of the unmodified ABI, unmodified applications run on these ports.

The Xen platform consists of a Xen VMM (virtual machine monitor) layer above the physical hardware. This layer provides virtual hardware interfaces to a number of Domains. These domains are effectively virtual machines running the ported guest OSs, however the guest OSs, and their device drivers, are aware that they are running on Xen. The Xen VMM is designed to be as simple as possible, so although it must be involved in data-path aspects (such as CPU scheduling between domains, data block access control etc.) it does not need to be aware of higher level issues, such as how the CPU should be scheduled. For this reason, the policy (as opposed to the mechanism) is separated from the VMM, and run in a special domain, called *Domain 0*. Domain 0 is given special privileges. For example, it has the ability

to start and stop other domains, and is responsible for building the memory structure and initial register state of a new domain. This significantly reduces the complexity of the VMM, and prevents the need for additional bootstrapping code in ported guest OSs.

There are two ways in which the Xen VMM and the overlying domains can communicate - synchronous calls from a domain to Xen can be made using *hypercalls*, and asynchronous notifications can be sent from Xen to an individual domain using *events*. Hypercalls made from a domain perform a synchronous trap to the Xen VMM, and are used to perform privileged operations. For example, a guest OS could perform a hypercall to request a set of page table updates. The Xen VMM validates these updates, to make sure the guest OS is allowed to access the memory requested, and then performs the requested update. These hypercalls are analogous to system calls in conventional operating systems. So that each hypercall does not involve a change in address space, with the associated TLB flush and page misses this would entail, the first 64MB of each domain's virtual memory is mapped onto the Xen VMM code.

Asynchronous events allow Xen to communicate with individual domains. This is a lightweight notification system which replaces the traditional device interrupt mechanism used by traditional OSs to receive notifications from hardware. Device interrupts are caught by the Xen VMM, which then performs the minimum amount of work necessary to buffer any data and determine the specific domain that should be informed. This domain is then informed using an asynchronous event. This specific domain is not scheduled immediately, but is informed of the event when it is next scheduled by the VMM. At this time, the device driver on the guest OS performs any necessary processing. This approach limits the crosstalk between domains, i.e. a domain will not have its scheduled time *eaten* into by another domain servicing device interrupts.

Xen provides a "Safe Hardware Interface" for virtual machine device I/O. This interface is discussed in detail by Fraser et al [13]. Domains can communicate with devices in two ways. They can either use a legacy interface, or an idealised, unified device interface.

The legacy interface allows domains to use legacy device drivers. Domains, however, cannot share devices to enforce isolation between domains. If a legacy device driver crashes, it will affect the domain it is running in, but not any other domains.

The unified device interface provides an idealised hardware interface for each class of devices. This is intended to reduce the cost of porting device drivers to this safe hardware interface. The benefit of this safe hardware interface is that it allows sharing of devices between domains. To enforce isolation between domains, even when they share a device, device drivers can be run within Isolated Driver Domains (IDDs). These IDD are effectively isolated virtual machines, loaded with the appropriate device driver. If a device driver crashes, the IDD will be affected, but each domain which shares that driver will not crash. In fact, Xen can detect driver failures, and restart the affected IDD, providing minimal disturbance to domains using that device.

Guest operating systems communicate with drivers, running within IDDs, using device channels. These channels use shared memory descriptor rings to transfer data between the IDD and domains. Two pairs of producer, consumer indices are placed around the ring, one for data transfer between the domain and the IDD, and the other for data transfer between the IDD and the domain. The use of shared memory descriptor rings avoids the need for an extra data copy between the IDD and domain address space.

[poss pic of descriptor ring]

Although the use of these descriptor rings is a suitable approach for low latency devices, it does not scale to high bandwidth devices. DMA capable devices can transfer data directly into memory, but if a device is shared, and the demultiplexing between domains is not performed in hardware, then there is

no way of knowing into which domain's memory the data should be transferred. In this case, the data is transferred to some memory controlled by Xen. Once I/O has been demultiplexed, Xen knows to which domain this data should be transferred, but the data is in an address space which is not controlled by that domain. The data is mapped into the correct domain's address space, using a page previously granted by this domain (i.e. this domain's page table is updated, so that the domain's granted virtual page now points to the physical memory which contains the data to be transferred). This page table update avoids any additional data copying due to the virtualisation layer.

The use of these techniques means that Xen has a very low overhead compared with other virtualisation technologies. The performance of Xen was investigated in [1], which, in a number of benchmarks, found that the average virtualisation overhead of Xen was less than 5%, compared with the standard Linux operating system. This contrasts with overheads of up to 90% in other virtualisation systems such as VMware and User-Mode Linux. These results have been repeated by a team of research students in [10].

3.4 Similar Work

There have been a number of efforts which have attempted to produce QoS-aware networks with some form of resource partitioning. Many of these projects use programmable networks. Programmable networks [4] can be used to rapidly create, manage and deploy network services based on the demands of users. There are many different levels of network programmability, directed by two main trends - the Open Signalling approach and the Active Networks approach. The Open Signalling approach argues for network switches and routers to be opened up with a set of open programming interfaces. This approach is similar to that of telecommunication services, with services being set up, used, then pulled down. The Active Network community supports a more dynamic deployment of services. So called, "active packets" are transported by active networks, which at one extreme, could contain code to be executed by switches and routers as the packet traverses the network.

A number of research projects have created programmable networks, in an attempt to support differing objectives. Many of these programmable networks have been created in an attempt to ease network management, however, some have had the objective of creating Quality of Service aware networks. The Darwin project [7] is an attempt to create a set of customisable resource management mechanisms that can support value added services, such as video and voice data streams, which have specific QoS requirements. This project is focused on creating a network resource hierarchy which can be used to provide various levels of network service to different data streams, depending on their requirements. The Architecture contains two main components - Xena and Beagle. Xena is a service broker, which allows components to discover resources, and identify resources needed to meet an application's requirements. Beagle is a signalling protocol, used to allocate resources by contacting the owners of those resources.

The Darwin architecture has been implemented for routers running FreeBSD and NetBSD. However this project's focus is on providing a middleware environment for value added network services, not on the underlying mechanisms needed to guarantee quality of service flows within a network. As such, the prototype implementation of the Darwin router simplifies the aspect of resource partitioning between QoS flows. Routing resources are managed by delegates, which run within a Java Virtual Machine with resource partitioning managed using static priorities. This does not provide the required level of performance, or real time guarantees required for a truly QoS aware router.

Members of the DARPA active networking program have developed a router operating system called NodeOS. This provides a node kernel interface for all routers within an active network. The NodeOS interface defines a domain as an abstraction which supports accounting and scheduling of the resources

needed for a particular flow. Each domain contains the following set of resources - an input channel, an output channel, a memory pool and a thread pool. When traffic from a certain flow enters the router, it consumes allocated network bandwidth from its domain's input and output channels. CPU cycles and memory usage are also charged to the domain's thread and memory pools as the packet is processed by the router. This framework allows resources to be allocated to QoS flows as required, however the NodeOS platform is simply a kernel interface, and does not provide details on how the resources used by domains should be partitioned.

The Genesis Kernel [5] is another network node operating system which supports active networks. This kernel supports *spawning* networks, which automate creation, deployment and management of network architectures, "on-the-fly". The Genesis Kernel supports the automatic spawning of *routlets*. Virtual overlay networks can be set up over the top of the physical network. Routlets are spawned to process the traffic from these virtual networks. Further child virtual networks can be spawned above these virtual networks. These child networks will automatically inherit the architecture of their parent networks, thus creating a hierarchy of function from parent to child networks.

This architecture is created with the intention of automating the support of network management. It supports a virtual network life cycle, through the stages of network profiling to capture the "blueprint" of a virtual network architecture, the spawning of that virtual network, and finally the management to support the network. It does not, however, deal with resource management, and so does not provide service guarantees to QoS flows.

The Sago project [9] has a similar aim to this project. The aim is to virtualise the resources of a network, to provide QoS guarantees for individual flows. The Sago platform uses virtual overlay networks to provide protection between separate networks which use the same physical hardware. It consists of two main components - a global resource manager (GRM), and local packet processing engine (LPPE). The GRM oversees the allocation of nodes and links on the underlying network. A LPPE is placed at each network node, and performs the actual resource management. The GRM uses a separate administrative network to signal the creation of virtual overlay networks. These virtual overlay networks have bandwidth and delay characteristics, which can be used to provide end to end QoS guarantees.

The Sago platform, however, requires significant extra complexity compared with standard networks. For example, two physical networks are needed, one for data transmission, and an entirely separate network to deal with control signals. This requires significant extra complexity in the network, increasing the overall cost and the possibility of failures. Also, although Sago attempts to provide end-to-end QoS guarantees, it does not deal specifically with the partitioning of resources within a network router.

Although there are many efforts which have attempted to provide QoS guarantees to network flows, very few have specifically dealt with the partitioning of resources among QoS flows. Of the efforts which did deal with router resources, none could provide strict guarantees on the router resources available to network flows.

4 Proposed Approach

The project goal is to produce an experimental router which demonstrates the feasibility of using virtual machine techniques to partition router resources between independent flows, thus providing guarantees on each flow's quality of service. It is not necessary that this router exhibit commercial speed or robustness, however it should be sufficiently usable such that the partitioning between flows and the service provided to those flows can be investigated. As such, my proposed approach involves using the work

of open source projects, modified so that they meet the needs of this project. Due to hardware and time constraints on a research project such as this, the router will be created as a software router for commodity x86 hardware.

The overall architecture of the router (known hereafter as a *QuaSAR* - Quality of Service Aware Router) will consist of multiple guest OSs running on top of a single virtual machine manager. Each of these guest OSs will run routing software to route incoming packets. There will be one main guest OS router, whose job is to route all of the best effort, non-QoS traffic. The rest of the guest OSs are known as *routelets*, and are available for the routing of QoS traffic flows. When a new QoS flow is initiated, the flow's requirements are sent along the proposed route of the flow. When a QuaSAR router receives details of a new flow's requirements, it first decides whether it has resources available to service the new flow. If there are insufficient resources available, a flow rejection message is sent. If, however, there are enough resources available, that flow is associated with a particular routelet, which is then given enough resources (CPU time, memory, network bandwidth, etc.) to service that flow. When a packet arrives at a QuaSAR router, it is demultiplexed and sent to the routelet which is dealing with that packet's flow. If the packet is not part of any specified flow, it is sent to the main best-effort router. Figure 3 gives an overview of the architecture of the QuaSAR system.

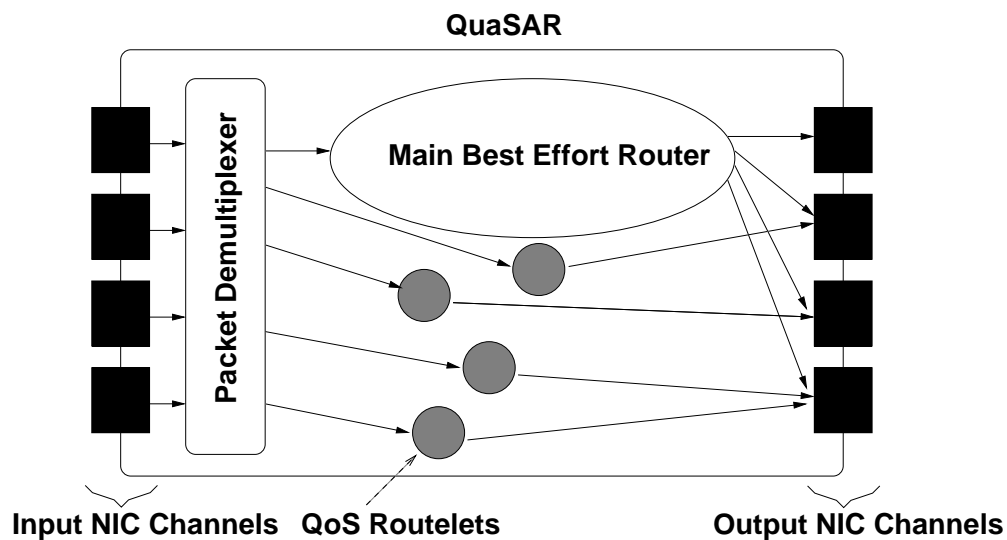


Figure 3: An overview of the QuaSAR architecture.

Since each routelet is running in a separate operating system (to partition its resource usage) QuaSAR will use virtualisation software, so that multiple operating system instances can be run on the same machine. This virtualisation software needs to have as little overhead as possible, so that QuaSAR can cope with high-volume packet traffic demands. To this end, this project will use a para-virtualisation system rather than a fully emulated virtualisation system because performance and scalability are more important than full emulation of the original instruction set, especially where the hardware being used (x86) is inherently virtualisation-unfriendly. Both Xen and Denali provide para-virtualisation in an attempt to increase performance and scalability, however Denali does not have any well-used operating systems ported to their virtual instruction set, whereas Xen has ports for both Linux and NetBSD. This project will therefore use the Xen virtualisation software because: it is open source and can be modified

for the project's requirements; it has good performance and scalability; and the availability of Linux or NetBSD guest OSs allow the project to make use of pre-built routing software.

Xen loads guest OSs in separate domains. The guest OS in domain 0 has special privileges, such as being able to start new guest OS domains and configure the resources which they can use. The main best-effort router will be started in domain 0, so that this main router can start, stop and control the QoS routelet instances. RSVP messages will be used to signal the creation of a new QoS flow by the hosts. This RSVP message will contain a flowspec, which details the requirements of this new flow (e.g. bandwidth, latency etc.). When the QuaSAR router receives an RSVP `RESV` message, the message will be processed by the main router. The flowspec will be checked to decide whether the router has the necessary resources (taking into account the other QoS flows currently being serviced by the QuaSAR router). If the QuaSAR router can meet this flow's requirements, then the main router in domain 0 will start a new routelet, or modify an existing routelet, so that it routes the flow's packets correctly, and so that it has enough resources to meet the flow's QoS requirements. Rather than starting a new guest OS instance for each flow, a *pool* of idle routelets could be started initially, and modified as needed.

When packets arrive at the router, they need to be demultiplexed to the appropriate routelet. Xen contains a mechanism for sharing network cards, and demultiplexing packets to the appropriate guest OS according to a set of rules. These rules can be changed by the guest OS in domain 0. The main router in domain 0 can therefore edit these rules so that packets are sent to the routelet which deals with its flow. This demultiplexing portion of Xen may need to be rewritten to cope with my requirements; however, it provides a good starting foundation.

QuaSAR will route MPLS (Multi-Protocol Label Switching) traffic, since this traffic is already routed as flow traffic by the forwarding equivalency class label. The main best-effort router needs to route any traffic which is sent to it. It therefore needs to create, maintain and use label and routing tables. It also needs to be able to support MPLS label distribution protocols (e.g. LDP or RSVP-TE), routing distribution protocols (e.g. OSPF or BGP), as well as the RSVP protocol used to create new QoS flows. The main router will use an open source implementation of an MPLS Linux router from IBCN (intec broadband communication networks) research group at Ghent University. This will need to be modified so that it can create and configure QoS routelets as required, but provides a solid basis from which to build the router without writing a completely new router.

The QoS routelets do not have complex routing requirements. They are only routing packets from a single flow. The work simply involves obtaining a new packet from the flow's input NIC, processing the packet (e.g. substituting one label for another), queuing the packet, and sending the packet to the flow's output NIC. Since only one flow is ever routed by these routelets, they do not need to maintain label or routing tables. There is also no need for them to understand, or initiate in routing or label distribution protocols. In fact, if routelets initiated distribution protocols, they could interfere with the main *domain 0* router. Instead, the main router will keep track of table entries which affect routelets that have been created. If one of these entries changes, then the routelet in question will be informed.

Due to the simple requirements of the QoS routelets, they will use the Click Modular Router to route packets. The simple work-flow needed by the routelets can be provided by joining a number of Click elements together. New elements can be written in C++ to any extra functionality required, such as MPLS label substitution. The main router can communicate with the routelets by writing to special files which are created by the Click architecture. When these files are written to, functions within the appropriate element are invoked.

Figure 4 gives a more detailed view of the proposed architecture.

Once the prototype QuaSAR system has been built, an important part of this project will involve exper-

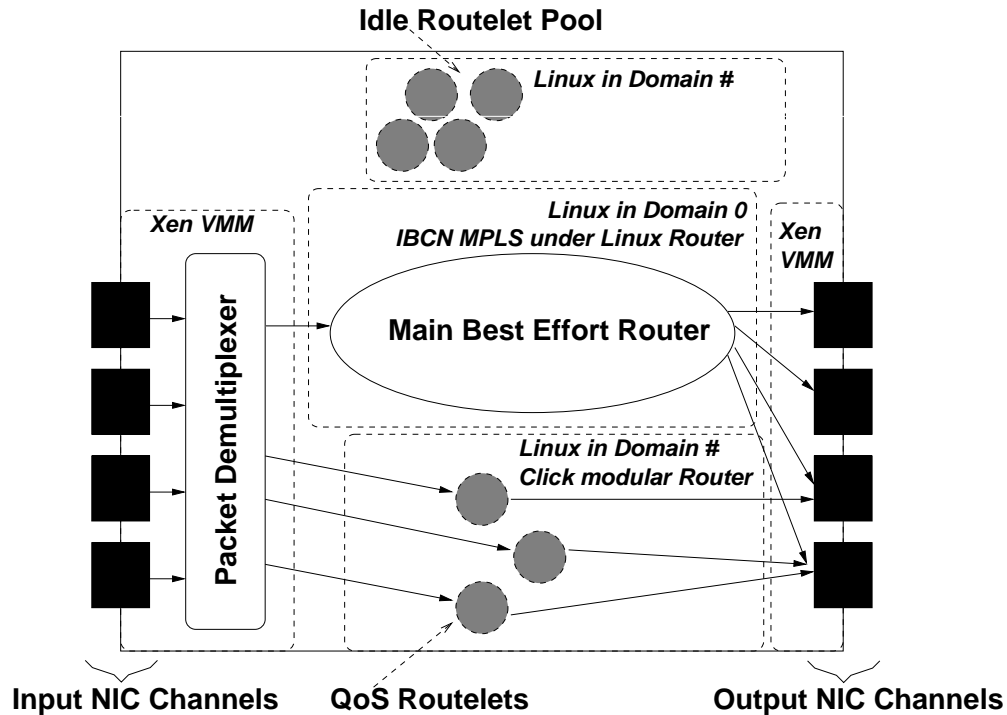


Figure 4: The proposed QuaSAR architecture.

imenting with this prototype to evaluate the effectiveness of the virtual routelet approach for QoS flows. This will entail ensuring that the virtualisation process does not massively reduce the performance of the router compared with standard routers, and discovering if this technique improves the performance and partitioning between QoS flows.

To this end, a testbed network will be set up, with a number of computers acting as communication nodes and one machine acting as a QuaSAR router. The QuaSAR machine will contain multiple network line cards, each of which will connect to a separate test network. Synthesised test network traffic will be sent across the QuaSAR router between these networks, to test various aspects of QuaSAR's performance. The machine running the QuaSAR software will also have a standard Linux software router installed, so that QuaSAR's performance can be compared with that of a standard router.

Initially, best effort traffic will be sent across both the QuaSAR router and the standard router to discover the added overhead incurred by running virtualisation software below the router. The next stage in evaluation will involve setting up a Quality of Service flow, and comparing the performance of traffic sent through this flow, to that of best effort traffic. This will investigate whether demultiplexing packets to different routelets incurs significant overhead, or if the simplified forwarding engine in the QoS routelets actually increases performance. The final stage will involve running multiple QoS streams and best effort traffic together to discover the interactions between QoS flows and between QoS flows and other traffic. This will allow the scalability of the QuaSAR router to be investigated, and the effectiveness of its flow partitioning. Badly-behaved flows, which use more network bandwidth than they have reserved, could be introduced to discover their effect on well-behaved flows, and evaluate QuaSAR's immunity to denial of service attacks. This needs to be investigated with badly-behaved and well-behaved flows using the same NICs (where saturation of the NIC's network bandwidth would become an issue) and with badly-behaved and well-behaved flows using the different NICs (where the CPU utilisation of the

baddly-behaved flow’s routelet would become more of an issue).

Once this evaluation has been performed on the QuaSAR prototype, it will be evident whether the approach posed by this project, of ensuring QoS by partitioning routers’ resources, is effective. It should also identify the areas in which this concept can be improved with future work.

5 Plan of Work

A week-by-week proposed plan of work for this project:

Week Beginning	Planned Work
20/12/04	Familiarisation with Xen, Linux, and Software Router technologies and source code. Setup of router machine environment. Holiday.
27/12/04	Familiarisation with Xen, Linux, and Software Router technologies and source code. Setup of router machine environment. Holiday.
03/01/05	Setup of Xen architecture with guest OS pool for routelets.
10/01/05	Creation of Click routelet.
17/01/05	Creation of Click routelet.
24/01/05	Automated configuration of routelet resources using Xen.
31/01/05	Automated configuration of routelet resources using Xen.
07/02/05	Generation of router resource requirements from RSVP flowspec.
14/02/05	Routelet configuration using RSVP path reservations.
21/02/05	Routelet configuration using RSVP path reservations.
28/02/05	Finish QuaSAR router.
07/03/05	Setup network performance testbed.
14/03/05	QuaSAR performance evaluation - comparison with software routers.
21/03/05	QuaSAR performance evaluation - flow partitioning.
28/03/05	QuaSAR performance evaluation - finalisation. Project report write-up
04/04/05	Project report write-up.
11/04/05	Project report write-up.
18/04/05	Finish project report & Hand in.

Figure 5 shows this information in a gantt chart.

References

- [1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, 2003, pp. 164–177.

This paper presents Xen, an x86 virtual machine monitor. The goal of this project was to support up to 100 virtual machines simultaneously on commodity hardware. Xen, therefore, is a para-virtualisation system, meaning that it presents an idealised machine abstraction, rather than attempting to fully emulate the underlying architecture. Operating systems must be ported to this idealised machine interface, however applications can be run on these ported guest operating systems without being ported themselves.

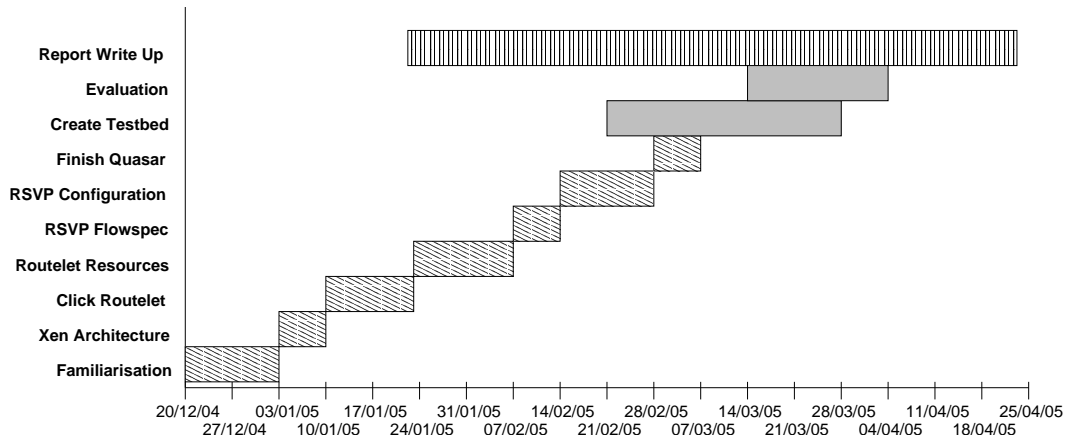


Figure 5: Gantt Chart of proposed work plan.

The paper first describes the idealised virtual machine interface. It then describes, in detail, the design of the Xen virtual machine monitor, and how it interacts with the virtual machine instances. The final section of this paper evaluates the performance of Xen compared with other virtualisation systems, showing the reduced overhead of para-virtualisation compared with full emulation.

- [2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, “RFC 2475: An Architecture for Differentiated Services,” Dec. 1998, Accessed: December 2004. [Online]. Available: <http://www.faqs.org/rfcs/rfc2475.html>

This RFC defines a QoS architecture which provides service differentiation in the Internet. The DiffServ architecture uses the DS field in IP headers to classify the QoS required by individual packets. Network nodes use different forwarding implementations depending on the packets class, to realise the required per-hop behaviours. This paper describes the Differentiated Services model, defining classifiers, traffic profiles, and traffic conditioners. It also gives some per-hop behaviour guidelines, and discusses Diffserv’s interoperability with non-Diffserv compliant nodes.

- [3] R. Braden, D. Clark, and S. Shenker, “RFC 1633: Integrated services in the Internet architecture: an overview,” June 1994, Accessed: November 2004. [Online]. Available: <http://rfc.sunsite.dk/rfc/rfc1633.html>

This RFC discusses how integrated services could be used in the Internet. It identifies a number of reasons why resource reservation could be useful in Internet traffic. It then describes the Integrated Service model which allows each individual application to reserve the resources it requires on the network. the basic functions of such a network service are described, including packet scheduling, packet dropping, packet classification, admission control and resource reservation. The paper then describes an example implementation of this architecture.

- [4] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, “A survey of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 2, pp. 7–23, 1999.

This paper provides a survey of the various types of programmable networks. It presents a model which provides a common framework for understanding different types of programmable networks. It first describes two different fields of thought in programmable

networks - open signalling networks, and active networks. The authors then describe a model which can be used to describe a programmable network, distinguishing the computation model used to support the programmable network, and the communication model used to distribute information. A number of characteristics of programmable networks are identified and described. These are used to classify various programmable networks.

- [5] A. Campbell, H. De Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela, "The Genesis Kernel: a virtual network operating system for spawning network architectures," in *Second International Conference on Open Architectures and Network Programming (OPENARCH)*, New York, March 1999, pp. 115 – 127.

This paper describes the Genesis Kernel, network node operating system which supports active networks. This kernel supports spawning networks, which automate creation, deployment and management of network architectures, "on-the-fly". The Genesis Kernel supports the automatic spawning of routlets which process traffic from virtual overlay networks. Genesis supports nesting of network functions, so a child network will have the capabilities of the parent network automatically. The paper discusses the overall architecture of the spawning Kernel, including the life cycle of the spawned virtual networks. It goes on to describe the programming environment used to create and manage spawned networks.

- [6] M. Carson and M. Zink, "NIST switch: a platform for research on quality-of-service routing," *Internet Routing and Quality of Service*, vol. 3529, no. 1, pp. 44–52, 1998.

This paper describes the NIST switch - a research router designed to support experimentation in the routing of QoS traffic flows. It is based upon MPLS, using labels internally for queueing and traffic shaping measures, and externally to identify paths over which traffic flows. This paper discusses the architecture of the NIST Switch. It proceeds to describe how the NIST Switch deals with label switching, and label distribution. The creation of explicit routes through the use of RSVP is also discussed. Finally the algorithm used to create label switched paths through the network is described.

- [7] P. Chandra, A. Fisher, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: customizable resource management for value-added network services," in *Sixth International Conference on Network Protocols*, Austin, October 1998, pp. 177 – 188.

This paper discusses the Darwin network architecture. This architecture consists of a set of customisable resource management mechanisms that can support value added services, such as video and voice data streams, which have specific QoS requirements. It creates a resource hierarchy of a network, allowing applications which have specific requirements to identify resources which can be used to meet these requirements. Darwin consists of two main parts: Xena, a service broker which discovers resources and identifies the resources needed by an application's requirements; and Beagle, a signalling protocol used to contact the owners of resources. This paper describes the design and implementation of Darwin, and provides some evaluation of the performance of the Darwin architecture.

- [8] Cisco, *Cisco IOS Switching Services Configuration Guide, Release 12.2*. Cisco Systems Inc, 2003, ch. 6, pp. 123–185.

This chapter gives an overview of the Multiprotocol Label Switching (MPLS) distribution protocol. It discusses how MPLS combines the performance of Layer 2 switching, with the scalability of Layer 3 routing. The paper gives details of how labels are distributed in an MPLS domain and how MPLS uses these labels to route packets. It then discusses some more advanced features, such as Traffic Engineering, VPN services and the MPLS class of service. Since this book is written by Cisco, the description of MPLS given is very biased towards Cisco's implementation of MPLS, and so is less useful as a general MPLS reference.

- [9] T. cker Chiueh, "Resource virtualization techniques for wide-area overlay networks," Computer Science Department, State University of New York at Stony Brook, Tech. Rep., 2003.

This paper presents "Sago" as a resource allocation algorithm. This algorithm takes network topology, dynamic traffic demands and resource scheduling into account. It also includes network wide fault tolerance. The network infrastructure uses virtual overlay networks to provide protection between separate networks which use the same physical hardware. However the system is severely complicated by the need for two separate networks, a control-plane and a data-plane. It also breaks the end-to-end argument by attempting to provide network wide fault tolerance.

- [10] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews, "Xen and the art of repeated research," in *Usenix annual technical conference*, 2004.

This paper attempts to repeat the results received at by "Xen and the art of virtualization". Overall, the authors were able to repeat the claims made by the original Xen paper. They found that Xen could easily support 16 moderately loaded servers on a single x86 machine. They also proved that Xen could run effectively on older machines, and even outperformed a IBM Z-series mainframe which was specifically built to provide virtualisation.

- [11] D. Decasper, Z. Dittia, G. M. Parulkar, and B. Plattner, "Router plugins: A software architecture for next generation routers," in *SIGCOMM*, 1998, pp. 229–240.

This paper discusses a high performance modular router which uses software plugins to allow for dynamic software upgrades of the router. The router can dynamically bind certain plugins to certain flows, allowing packets from different flows to have different operations applied to them as they pass through the router. This allows distinct plugins which perform the same basic function (e.g. priority scheduling and round robin scheduling) to coexist in the same router at the same time. When a packet enters the router, it moves through the network protocol stack. As it does so, it hits "gates", which allow the flow of execution to branch off to a plugin instance. The actual plugin which is called is decided by an association identification unit, which classifies the packet as belonging to a certain flow and passes it to the appropriate plugin for that flow. The paper discusses filter algorithms which allow efficient packet classification and flow caches which further speed up the classification of recently seen flows. These optimisations allow the router to provide a modular router framework with only an 8% overhead compared with a monolithic routing kernel.

- [12] R. Fatoohi and R. Singh, "Performance of Zebra Routing Software," Computer Engineering, San Jose State University, Tech. Rep., 1999.

This paper compares the Zebra routing software with commercial Cisco routers. To do this, the author created a network testbed which was used to investigate how well both Zebra and Cisco routers can connect different networks. The interoperability of the routers was tested by connecting networks with different layer 2 protocols through the routers. A frame relay router was used to create a software configurable way of selecting which route data should take through the system. Zebra was found to connect the networks in the way expected, with similar interoperability to a Cisco router. The authors also praised Zebra for its ease of use, management support and routing features.

- [13] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," in *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS'04)*, October 2004.

This paper discusses how Xen achieves isolation between device drivers in a virtualised machine. Xen provides I/O spaces which provide unified interfaces to guest OSs for different classes of devices. Isolated driver domains (IDDs) effectively run device drivers in separate virtual machines to isolate them from guest OSs. The device drivers communicate with guest OSs through device channels. A IDD can have multiple channels to guest OSs, thus allowing sharing of devices between guest OSs while still providing a level of isolation between these OSs. The paper discusses a number of interesting techniques for virtualised hardware access.

- [14] P. V. Heuven, S. V. D. Berghe, J. Coppens, and P. Demeester, "RSVP-TE daemon for Diffserv over MPLS under Linux," <http://dsmppls.atlantis.rug.ac.be> accessed November 2004.

This paper describes a Linux daemon which supports MPLS forwarding. It supports the creation of label switched paths using RSVP. The paper describes the way in which this daemon was build, with various separate components merged and modified to form the overall networking daemon. Finally the overall architecture is described, with the packet processing flow between components detailed.

- [15] R. Keller, L. Ruf, A. Guindehi, and B. Plattner, "PromethOS: A dynamically extensible router architecture for active networks," in *IWAN 2002, Zurich, Switzerland, 2002*.

This paper discusses PromethOS, a modular router architecture which is based upon the Linux kernel. This router allows plugins to be dynamically installed and configured to allow the router to have different functionality. The paper also discusses explicitly routed paths, which allow a packet to be marked so that it traverses a certain route through the routers, rather than following a straight IP route. There are mechanisms which allow plugins to be installed along a route, allowing router functionality to be installed as required. This paper was based on the work by Decasper et al in creating router plugins.

- [16] K. Lai and M. Baker, "A performance comparison of UNIX operating systems on the pentium," in *USENIX Annual Technical Conference*, 1996, pp. 265–278.

This paper compares and contrasts the performance characteristics of three variants of UNIX - Linux, FreeBSD and Solaris. The authors used a number of free micro and application benchmarks to characterise the untuned behaviour of the operating systems. Four main areas were investigated, system call / context switch overhead, memory performance, file system performance and network performance. It was found that Linux

had the best performance on file system operations because of its asynchronous meta-data updates. FreeBSD had the best network performance for both raw, UDP and TCP streams. The performance of Solaris generally fell in between the other operating systems. The memory performance of all three systems was broadly similar, but fell short of the full underlying potential of the processor. This paper was of limited use because of its age, and because it relies heavily on micro-benchmarks rather than real world applications. It does however provide a broad overview of the relative merits of these flavours of UNIX.

- [17] C. Metz, "IP Routers: New Tools for Gigabit Networking," *IEEE Internet*, vol. 2, no. 6, pp. 4–18, November 1998.

This paper summarises the techniques used to build modern Gigabit IP routers. It discusses the main features a router should possess, such as speaking multiple protocols, and being able to create an internetwork with other routers. The paper proceeds to describe the basic architecture of a router, consisting of line cards, a router processor and a backplane. The evolution of this architecture is discussed, with improvements such as switching fabrics and improved table lookup algorithms presented. This paper gives a good overview of the basic features and architecture of a IP router.

- [18] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Symposium on Operating Systems Principles*, 1999, pp. 217–231.

This paper discusses a software architecture which allows new router configurations to be created quickly and easily. A click router is built up from a collection of packet processing modules called "elements". These elements are written to perform simple router functions, such as classification, queueing or network card interfacing. Elements can be instantiated and connected using a simple configuration language. Elements can be connected using ports, which can either push or pull packets as needed. It uses a flow based router context, which allows different elements to find each other, for example a RED element could ask for the queue elements which follow it in the packet flow. It would then be passed references to those queues which the RED algorithm is interested in. The authors created a standards-compliant IP Router using click to prove that it was effective. Since elements are compiled to machine code, the forwarding rate of the click IP router is about the same as that for a monolithic software router.

- [19] C. Partridge, "A 50-Gb/s IP Router," *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pp. 237–248, June 1998.

This paper discusses the design and implementation of a high performance 50Gb/s IP router. The main innovation in the router presented was the movement from a centralised routing processor to a number of independent forwarding engines. A centralised network processor kept these forwarding engines in sync. The router also used a switched bus as a backplane, rather than the shared bus conventionally used at that time. The router was not fully implemented at the time of the paper so no quantitative performance measurements are given, however the architecture presented has been integrated into most modern high performance routers.

- [20] L. Peterson, "NodeOS interface specification," Active Networks NodeOS Working Group," Technical Report, February 1999.

This paper defines the interface of NodeOS - an operating system to be run on each node, or router, of an active network. The NodeOS network router kernel interface allows routers to support programmable networks. The NodeOS interface defines five primary abstractions: thread pools, memory pools, channels, files and domains. The first four abstractions encapsulate the router's resources. The fifth abstraction, a domain, is responsible for accounting, scheduling and admission control of resources needed by network flows. Although this paper defines an interface for resource management in network routers, it does not specify how this resource management should be implemented

- [21] I. Pratt and K. Fraser, "Arsenic: A user-accessible gigabit ethernet interface," in *INFOCOM*, 2001, pp. 67–76.

This paper discusses the building of a user-accessible ethernet card called Arsenic. This interface implements multiplexing and protection mechanisms directly in the line card. Applications can then create virtual "interfaces" which are independent of other applications' "virtual interfaces", and allow packets to be sent without operating system intervention. I read this paper to investigate the possibility of demultiplexing packets to virtual routelets directly from hardware. As this paper is by the same authors as the Xen paper, the interfaces were similar to those provided in Xen IO.

- [22] A. Ramanath, "A study of the interaction in BGP/OSPF in Zebra/ZebOS/Quagga," Computer Science Department, State University of New York at Stony Brook, Tech. Rep., 2000.

This paper studies the implementation of the Border Gateway Protocol (BGP) in Zebra family of routers (which includes Zebra, ZebOS and Quagga). It gives a basic overview of the BGP, discussing how routing information is transferred between separate routers, and how this routing information is used to make decisions about the route to a host. The paper then gives a brief overview of Zebra's architecture, with emphasis on Zebra's technique of running each routing protocol (e.g. BGP or OSPF) in separate daemons in the Linux kernel. The author provides detailed information about how these protocol daemons, and Zebra as a whole, can be configured. The paper does not, however, seem to provide any information about the interaction between BGP and OSPF (Open Shortest Path First), which was stated as being one of the main aims of this paper.

- [23] G. Rosenbaum, S. Jha, and M. Hassan, "Empirical study of traffic trunking in Linux-based MPLS test-bed," *Int. J. Netw. Manag.*, vol. 13, no. 4, pp. 277–288, 2003.

This paper describes the creation of an MPLS testbed, and its use in an evaluation of the benefits of traffic trunking using MPLS. The paper first describes how the MPLS testbed was implemented. It then goes on to describe test cases which were used to evaluate the effect of traffic trunking in MPLS networks. Finally the paper discusses the way in which flows interact if they are mapped into the same trunk. For example, the authors find that UDP traffic heavily degrades TCP traffic, if both flows share the same traffic trunk. The paper concludes that the use of trunks in MPLS is effective at isolating flows and improving the perceived QoS in end systems, however, care should be taken when mapping traffic to trunks, if isolation is to be enforced.

- [24] J. Sugerma, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, 2001, pp. 1–14.

This paper discusses how VMware approached IO virtualisation on the x86. It discusses how guest OS instances can be shielded from each other when interacting with the same device. The techniques presented in this paper were focused on complete universal emulation of device drivers, and not on the performance limitations this emulation required. The techniques presented are therefore not appropriate for a high performance, specialised router.

- [25] The International Engineering Consortium, “Multiprotocol Label Switching (MPLS),” <http://www.iec.org/online/tutorials/mpls/> - Accessed November 2004.

This paper presents a general overview of the MPLS distribution protocol. It describes how forward equivalence classes (FECs) are set up, and how packets in these classes are routed through label-switched paths (LSPs). It also identifies the differences between label edge routers (LERs) and label switching routers (LSRs) and where these would be placed on the MPLS domain. It discusses issues such as label binding, creation, distribution, merging, retention and control. It also presents more advanced issues such as traffic engineering, tunneling and multicast operation. This paper gives a broad overview of the MPLS protocol, without going into many specific implementation issues.

- [26] A. Whitaker, M. Shaw, and S. Gribble, “Denali: Lightweight virtual machines for distributed and networked applications,” University of Washington, Tech. Rep., 2002.

This technical report discusses Denali, a virtual machine manager for network server applications. It was created to run specifically on inexpensive commodity x86 hardware. Denali uses para-virtualization techniques to increase the performance of the system, at the expense of guest OS portability. The authors discuss the implementation of Denali and Iiwaco, a simple guest OS which can be run on Denali. The authors present a number of experimental benchmarks which demonstrate the scalability of Denali and its network performance.

- [27] A. Whitaker, M. Shaw, and S. D. Gribble, “Scale and performance in the Denali isolation kernel,” *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 195–209, 2002.

This paper describes scale and performance issues in the Denali Isolation Kernel. Denali is a virtual machine manager, which has the goal of pushing infrequently used Internet services on to third party infrastructure. To do this, it must scale to possibly thousands of virtual machine instances. The authors do not try to fully emulate the x86 hardware in the virtual machines, but instead provide an abstraction, which simplifies the implementation and improves performance. A number of experimental results are presented which show that it would be possible for Denali to host 10,000 virtual machines, however there is a substantial cliff at about 1000-2000 VM instances which would severely reduce these VM servers’ throughput.

- [28] J. Wroclawski, “RFC 2210: The use of RSVP with IETF integrated services,” Sept. 1997, Accessed: November 2004. [Online]. Available: <http://www.faqs.org/rfcs/rfc2210.html>

This RFC discusses how the RSVP reservation protocol can be used to provide Quality of Service mechanisms through Integrated Services. It discusses a number of uses for RSVP in the Internet architecture, and gives a summary of its operation. It describes in detail a number of resource reservation objects which can be carried by RSVP, such as the: TSPEC object, which carries data about the source’s generated traffic; the flowspec

object, which describes the requirements being requested by the flow; and the ADSPEC object which can be used by services to transmit data to the application. While these objects are transmitted by RSVP, they are transparent to the RSVP protocol itself and so can be changed without requiring changes to the RSVP protocol.

- [29] X. Xiao and L. M. Ni, "Internet QoS: A big picture," *IEEE Network*, vol. 13, no. 2, pp. 8–18, March 1999.

This paper summarises the main techniques available for Internet Quality of Service. It discusses techniques such as integrated services, RSVP, differentiated services, multi-protocol label switching (MPLS) and constraint based routing. It compares these techniques, indicating their relative merits. Two likely end-to-end service architectures are described to illustrate where these techniques fit overall. The paper also compares ATM networks to router networks with differentiated services and MPLS, explaining the differences between virtual circuit switched networks and traffic engineered packet switched networks. The paper gives a good overview of the techniques available for providing QoS in the Internet and is useful for comparing these techniques.

- [30] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Network Magazine*, vol. 7, no. 5, pp. 8–18, September 1993.

This paper discusses a resource reservation protocol which can be used to extend the Internet's best effort service to support reserved Quality of Service flows. The paper discusses only the reservation protocol (RSVP), not the other elements such as flow specification, routing, admission control and packet scheduling which would be required to build a network supporting QoS dependent traffic. The distinguishing feature of this reservation protocol is that it is receiver-initiated. This allows for heterogeneity among receivers' reservation requests, as well as dynamic membership changes. It also decouples the reservation and routing functions. The paper presents the various features of RSVP with a number of simple, practical examples.

- [31] J. A. Zubairi, "An automated traffic engineering algorithm for MPLS-Diffserv domain," *Proc. Applied Telecommunication Symposium*, pp. 43–48, April 2002.

This paper discusses the use of Diffserv in an MPLS domain for traffic engineering. It describes a traffic colouring technique for engineering the traffic through a network and provides the required level of QoS. The authors implemented this traffic engineering technique in C++, and used this implementation to model the effectiveness of the system. They found that the algorithm performed in a very predictable way, allocating resources until a threshold where there is no longer any available bandwidth, at which point excess requests are denied. However, Diffserv defines classes of service in each individual packet, it does not set up an initial reserved flow for QoS dependent traffic. Since my project involves reserving resources on routers for QoS flow, the Diffserv model is not appropriate.