Script-driven Packet Marking for Quality of Service Support in Legacy Applications

Timothy Roscoe and Gene Bowen
Sprint Advanced Technology Labs
1 Adrian Court
Burlingame, CA 94010, USA
{troscoe,higene}@sprintlabs.com

1 ABSTRACT

This paper describes the implementation of a system to deliver Quality of Service for IP flows using a DiffServ-like packet marking mechanism. The system uses an unmodified commodity operating system (Windows NT), and a policy daemon is employed to implement arbitrary policies for QoS via a scripting mechanism. By interposing an agent in the protocol stack used by the application runtime system, off-the-shelf applications can have different packet forwarding policies assigned to different flows they originate, without any need to recompile either the operating system or the application. The principle of the system can be naturally extended to implement more widely coordinated policy-based networking, and network reservations using protocols such as RSVP, without any need to recompile applications.

2 INTRODUCTION AND MOTIVATION

This paper describes the implementation of a technique to allow existing applications to take advantage of Quality of Service (QoS) facilities provided by an IP-based network (specifically, the facilities described by the DiffServ [BBC+98] framework) and thereby improve their performance without modifying the application code, or the underlying operating system (in this case, Windows NT).

This paper does not provide quantitative measurements of the performance of the techniques it describes. Rather, the point of this paper is to illustrate the techniques themselves, argue for their usefulness, and describe a demonstration scenario which shows their applicability. We believe the ability to enhance existing applications to take advantage of network facilities to be important technically: even carefully written applications targeted at a best-effort network like the Internet perform poorly over long distances, particularly during periods of network congestion. Furthermore, such applications are frequently unable to take advantage of any improved levels of service that the network might be able to deliver, since they are incapable of indicating their needs to the network.

This is important from a business perspective as well. It has long been recognised that some classes of applications (particularly distributed multimedia applications) require QoS guarantees of some kind from networks and end-systems in order to deliver a compelling service to users. Indeed, this has inspired a large amount of academic research over the years into QoS support in operating systems (for example, [LMB+96, MP96]), QoS APIs and programming models (for example, [BSY+97]), so-called QoS Architectures (see [CCG+93, Nic90] and others), as well as the wealth of QoS work in the field of networking proper. Despite the sustained quality of this work, actual deployment of QoS-aware applications remains exceedingly rare, even in networks which have for some time explicitly provided QoS mechanisms (such as some forms of ATM). Commercial Internet applications which are capable of explicitly requesting resources are unheard of: one of the additional outcome of our work is a comprehensive view of how applications on Windows NT are invoking the operating system's networking facilities, and we have yet to encounter *any* applications which make use of the Microsoft Generic Quality of Service (GQOS) facility in Winsock 2.2.

The reason usually cited for this lack of deployment is a chicken-and-egg problem: network providers don't have any customers requesting different levels of service from the network, since those customers have no applications which can use such capabilities[1]. Consequently, network equipment vendors see no demand for such facilties in network elements (and the need for end-to-end deployment of these facilities, across potentially several different vendors and networks, further discourages implementation. Finally, software developers see to point in trying to make use of facilities which are not implemented in the network, and so adopt two, complementary tactics: applications do the best they can with a best-effort network, through techniques such as large buffers

[1] A debatable exception in this case might be the provision of IP Virtual Private Networks (VPNs), but these are not applications in the sense of the word as used in this paper.

to absorb jitter, layered coding, etc., and applications which might really take advantage of potential QoS features in the Internet (telemedicine, for example) simply don't get written in the first place.

The work described here was motivated by a desire to break this circle. The idea is that if existing applications can be shown to benefit from network functionality which delivers different levels of service, *without the need to change or redeploy these existing applications in any way*, there will be an incentive to implement QoS mechanisms in the network that application designers and software engineers can subsequently explicitly take advantage of.

3 DESIGN

The design of the system presented here has a number of goals:

No modification of programs: There should be no need to modify, or even recompile, any applications. Without this goal, we are back in the cycle described above preventing adoption: already deployed applications must be able to benefit from our work.

No modification of the operating system: A similar argument applies to changes to operating system code; use of non-standard operating systems (even with unchanged APIs) is too much of a disincentive to adopt QoS-enabled network facilities.

Encapsulation of policy: The policies which determine levels of network service granted to an application must be encapsulated in a central point, at least on a per-machine basis. The policies cannot live in the applications, since our assumption is that applications can know nothing of network QoS. By centralising policy we can maintain a clear picture of how a given machine is trading off different applications' requirements.

Policy expressiveness: It should be possible to express complex policies regarding how levels of service should be assigned, policies that include attention to local machine state, different human users, different remote endpoints for communication, etc. Policies based on simple lookup, for example in a directory service, are deemed to be insufficiently expressive.

The goal of modifying neither applications nor operating system limits us to installing additional software on the machine (and, ideally, making it possible to remove all trace of this additional software if need be). This led us to deploy a mechanism which interposes itself between the application and the operating system, intercepting network operations and inserting new ones.

Encapsulation of policy results in a design with a central process making all policy decisions. This centralisation allows us to coordinate per-machine state when making policy decisions on a per-application basis.

The requirement for expressiveness led us to employ a scripting language (SafeTcl in our case) to express policies. A scripting language permits an open-ended set of policies which can take into account al kinds of application attributes, as well as creating and maintaining local state to be used in making decisions.

3.1 The DiffServ framework

This work was carried out loosely in the framework of a DiffServ-capable network. DiffServ [BBC+98] is a proposed IETF model for offering differentiated services in the Internet. IP Packets are marked with a byte value known as the DS field [BBBN98] (formerly the Type of Service octet [Alm92]), which specifies how the packet should be treated on a per-hop basis by routers. Various Per-Hop Behaviours (PHBs) have been proposed for association with DS Field values.

It should be pointed out that DiffServ is primarily a facility to be used in the core of the network: traffic entering a provider's network is policed at the edge on a per-flow basis to ensure that it conforms to a service-level agreement (SLA) between the network provider and the source of the traffic (usually another network). Thereafter, the idea is that per-flow state in the network core's routers and switches is not required: they simply examine the DS field to determine per-hop behaviour, and the admission control procedure applied to the SLAs, together with the edge policing, should ensure that guarantees are met.

Traffic can be marked in the end-user's network either at an access router (where it leaves the user's network and enters the wider Internet), or in the end systems themselves. Our work addresses the latter case, but can be applied where the gateway router implements the marking policy itself as well.

4 IMPLEMENTATION

Our design principles lead to a system with three principal components:

1. A per-application protocol agent which is installed in the network protocol stack for each application, executes in the same address space as the application. The agent intercepts networking calls by the application, and can set a particular packet mark for each file descriptor opened by the application.

2. A per-machine QoS policy daemon which holds the policy information as to how packets from different network connections on the machine should be marked, and relays the implications of this policy to the protocol agents.

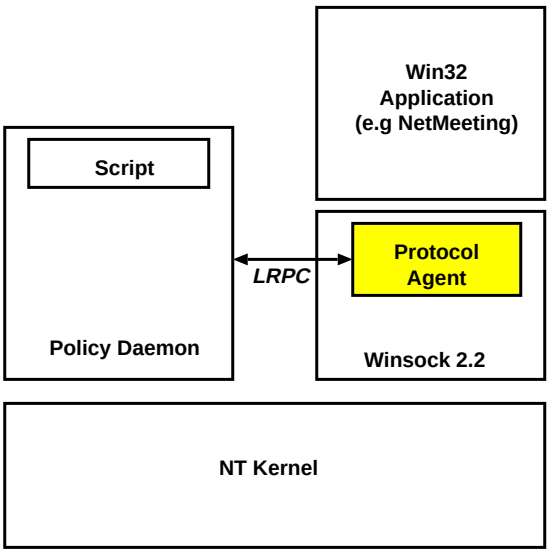

Figure 1: Components of the system

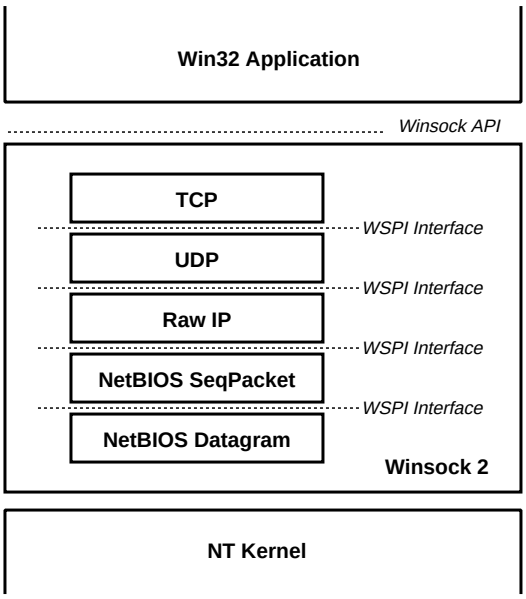

Figure 2: Layering of Winsock Service Providers

3. Networking elements capable of interpreting the packet markings appropriately. In practice these are IP switches and routers implementing diffserv-like mechanisms.

These relationship between these components is shown in figure 1. We describe each of them below.

4.1 Mechanism: The Protocol Agent

The basic mechanism we use to add some quality of service support for existing applications is to interpose a piece of software in each running application which intercepts calls made by the application to the networking facilities provided by the underlying operating system.

In Microsoft Windows NT[2], most applications are written using the "Win32 personality", which can be thought of as a large runtime subsystem between the kernel proper (as defined by the system call interface) and the application itself. How Win32 is actually implemented is beyond the scope of this paper; what we are interested in here is WinSock, the Windows networking API.

WinSock is structured as a set of layers, called "service providers" (see figure 2). Each service provider implements a standard interface, the "Winsock Service Provider Interface" or WSPI, which is invoked by the layer above it. The service provider in turn invokes operations on the SPI of the provider below it. At the top of the stack is the Winsock API proper, which is invoked by the application itself. At the bottom is the interface to the kernel's networking functionality. Note that the layering does not necessarily imply dependence of one service provider upon a lower one. Since all service providers execute in the same protection domain as the application, and most of the TCP/IP functionality is actually in the kernel for protection reasons, the service providers for protocols like TCP don't really contain the protocol implementation but instead are thin layers above the kernel interface.

New service providers, in the form of Dynamic Link Libraries (DLLs) can be installed between any two existing layers in the Winsock stack by entering their path names in the Windows Registry [HOB99]. This has the effect of causing all programs subsequently launched to include the new provider in the per-process protocol stack.

This gives us a way implement an agent to intercept networking calls made by off-the-shelf applications, and insert our own behaviour in the chain—the protocol agent is simply a new service provider which contacts the policy daemon (if present) whenever the application makes a Winsock call we are interested in, and otherwise passes all calls and returns straight through[3].

The full set of Winsock calls intercepted by the protocol agent is shown in figure 3, divided into those calls that actually refer to a socket descriptor, and those calls which deal with the initialisation and destruction of the library (Winsock) state itself when the application starts up and exits respective-

[2] We use Windows NT version 4.0 (Workstation or Client versions), with Service Pack 4 installed. Because what we do is entirely within WinSock, our system should also work with Windows 98, though we have not tried this

[3] In practice, the need to support asynchronous I/O considerably complicates the implementation of even an "null" service provider, since state must be maintained at all layers. However, our agent only adds a few hundred lines of code to the simplest, minimal service provider.

Call	Description
`Socket`	Allocation of a socket
`Close`	Closing a socket
`Connect`	Connect a socket
`JoinLeaf`	`Connect` for a multicast group
`Listen`	Wait for connections on a socket
`RecvDisconnect`	Receive disconnect
`SendDisconnect`	Send disconnect
`Shutdown`	Shutting down a socket
`Startup`	Library initialises
`Cleanup`	Library finally shutting down

Figure 3: Winsock 2.2 calls intercepted by the agent

```
[
   uuid(97cd437c-9b57-11d2-b8c7-00c04f79ebc1),
   version(1.0),
   endpoint("ncalrpc:[sprint_pold]")
]

interface PolicyDaemon
{

   [string] char *Event( [in, string] char *event );
}
```

Figure 4: Microsoft Interface Definition Language (MIDL) interface to the policy daemon

ly.

4.1.1 Initialisation and Shutdown of the Agent

As a Winsock service provider, the agent's `Startup` function is called when an application loads. Aside from the usual service provider initialisation functions, the method attempts to establish a local RPC (LRPC) binding to the policy daemon running on the machine. Assuming that a connection to the policy daemon is successfully established, the agent sends a short message to the daemon indicating its presence.

As figure 3 indicates, we also send notification when the `Cleanup` method on the service provider is called. In theory, this occurs when the library is unloaded, and gives the policy agent some information which can be used to update its state as to which applications are still executing. In practice, we find that most applications do not shut the Winsock system down gracefully at all, and so this method is almost never invoked. Under normal circumstances this doesn't cause serious problems; Winsock is only a set of shared libraries, after all. Machine resources used by any active network connections (sockets, for instance) are kernel resources and are cleaned by NT when the process exits. However, for our purposes it prevents the policy daemon from reliably knowing when an application ceases using the network, which is unfortunate.

4.1.2 Communication Transport

The choice of LRPC as a communications transport between agents and the policy daemon was motivated by a number of factors. Firstly, we naturally wanted to avoid any network communications that might themselves use the Winsock stack. In practice this is not as serious a restriction as it might be, since there are Win32 network communication facilities that bypass Winsock and simply invoke the NT kernel services, However, keeping everything explicitly local keeps things simple, and should allow us to run the same code on Windows 98.

Secondly, we wanted a fairly efficient mechanism. Since an invocation on the policy daemon will be made for every Winsock call in table 3 by every Win32 application on the machine, imposing a high overhead is to be avoided. However, this is also less serious than it might sound: we don't intercept the `Read` and `Write` calls that are likely to be most heavily used, and so placing our agent in the IP equivalent of the network control plane is much less of an overhead than being on the data path. In practice we have not experienced any perceptible degredation in performance of applications.

Robustness is a important goal of the protocol agent, especially since it is introduced into every instance of the Winsock stack running on the machine. For this reason, it's important that the agent not rely on the presence of an executing policy daemon on the machine for correct operation; if it did, the failure of our daemon would at best cut the machine off from the network, and at worst prevent any Win32 applications from functioning. Our implementation catches any exceptions from the bind process and initial message transmission, if there is a failure of any kind indicating that the policy daemon cannot be contacted, the agent sets a flag accordingly and gives up—all subsequent Winsock operations continue as before.

All communication between between the agents and policy daemon consists of an exchange of textual strings; the RPC interface to the daemon is therefore extremely simple (see figure 4).

We might have chosen to define the signature of every operation more rigorously using the type system provided by Microsoft's IDL, but we made a conscious decision not to do this. Instead, the arguments are marshaled into a list of tokens according to the lexical rules of the scripting language used in the policy daemon, in our case Safe Tcl [OLW97, Wel97]. The code to correctly format and escape our arguments is quite simple and coded into the agent: we don't employ the Tcl library functions themselves so as to avoid hauling the Tcl library in its entirety into the application, which would result in reduced robustness and the possi-

bility of symbol conflicts with applications using a different version of the library, or multithreaded programs potentially accessing global variables in the library at the same time as us. It's important in this situation to know where we stand with the agent code.

Since the agent and the protocol daemon are very closely coupled, and invocations of this kind only occur between them, the advantage of specifying the interface more fully in MIDL would be minimal. Furthermore, the code we would need to write to extract the arguments to Winsock calls and place them in data structures mandated by the MIDL language mapping is of comparable complexity with the string marshaling code we use anyway. Finally, having the arguments in Tcl form greatly simplifies the daemon implementation, described below.

4.1.3 Runtime Behaviour

For each of the runtime calls in figure 3, the protocol agents invoke the policy daemon and interpret the result that is sent back. The string passed with the invocation, considered as a sequence of Tcl tokens, always starts with four standard elements:

1. The type of the message. In almost all cases, this is the name of the Service Provider Interface method which has been invoked. The exception is one additional message type for logging debugging information from the agent DLL to the policy daemon.
2. The process ID of the NT process within which the application is executing.
3. The NT username under which the application is executing
4. The command line used to invoke the application. This is present even when the application is launched from the Windows graphical shell, and so allows us to distinguish different application classes easily, as well as see the arguments they were invoked with.

Following these elements are typically the arguments to the SPI method. These include such useful information as addresses to connect to and socket identifiers, as well as QoS requests to the operating system specified using GQOS, Microsoft's QoS extensions to the networking API. Unfortunately, we have yet to see an application which supplies the additional GQOS parameters to Winsock, so such values are generally null.

The SPI method arguments, together with the initial information about the application currently running, provide enough data to the policy daemon for it to make decisions as to how packets should be marked for each connection. The string returned from the daemon is used to convey this information. A present, the string is either empty or contains a string of the form "`TOS` textitn", where *n* is the value for the Type-of-Service octet ([Alm92], now known as the DiffServ field, [BBBN98]) in the IP header. This can be set using the appropriate `setsockopt` call to the kernel.

Extensions to this trivial return syntax are possible, most compelling is the addition or modification of GQOS specifications on the socket. This would allow us to assign rather more specific levels of service to individual applications than simple DiffServ allows, for example using RSVP [BBH+97] "behind the back" of the application to reserve network resources for audio or video streams, for instance.

Alternatively, a different response could prevent the method from completing successfully and instead make it return an error, allowing the system to perform some measure of admission control and reject network operations if they would result in serious degradation of service for multimedia applications already running on the machine.

Two calls which we do not attempt to intercept in the protocol agent are `read` and `write`. We don't see much advantage to be gained from this, other than the gathering of statistics on application behaviour, especially since traffic shaping facilities are now available the Windows NT kernel as well as in Linux and other Unix-like systems. Furthermore, the cost of a local RPC call for each network read or write is almost certainly too much, particularly for a time-senstive multimedia application that cannot afford that many context switches. Finally, this would probably result in the policy daemon itself becoming a bottleneck.

4.2 Policy: The QoS Policy Daemon

The function of the QoS Policy Daemon is to assign different packet markings to network connections originating at the machine, based on some policy. In a sense, the daemon is where the policy is encapsulated as far as the applications (each of which now includes a protocol agent) are concerned.

The implementation of the daemon is very simple - essentially, invocations from the protocol agents executing inside applications arrive as well-formed Tcl commands (since the first token in the string is the message type), and are simple executed by a SafeTcl interpreter.

4.2.1 Quality of Service scripts by example

At startup, the interpreter is loaded with a script which encodes the policy set for the machine. Figure 5 shows a simple script, which on our NT machines assigns a higher importance to network connections initiated by an instance of NetMeeting running on the machine. The script consists of only one Tcl procedure, `Socket`. This procedure is invoked when any Winsock client creates a socket, *after* the call to the lower layer service providers has completed. This explains the additional "`fd`" argument—it is the socket descriptor which has been allocated.

```
#
# Give NetMeeting better network service
#

# Read the helper functions
source SprintLib.tcl

# String to recognize NetMeeting on this machine
set NetMeetingStr
    "\"C:\\Program Files\\NetMeetingNT\\conf.exe\" "

# Intercept Socket calls and set packet mark
proc Socket { pid uid cmd family type protocol fd } {

   global NetMeetingStr AF SockType

   if {[string compare $cmd $NetMeetingStr] == 0} {
      if { $family == $AF(INET)
                && $type == $SockType(DGRAM) } {
         # High priority to NetMeeting video/audio
         return "TOS 48"
      } else {
         # Give some help to TCP (for signalling)
         return "TOS 16"
      }
   }
}
```

Figure 5: Example script to deliver better service to NetMeeting streams (slightly reformatted for clarity)

The script makes use of a couple of arrays defined in the file of helper functions "`SprintLib.tcl`" which contain mnemonic names for socket type names and address families, making it easier to read the script as well as improving portability to systems with different numeric assignments for these values (all such values are passed numerically across the LRPC connection).

If the call in question is from an application whose command line matches NetMeetings, then we assign a Type-of-Service value to 48 to Internet datagram connections (i.e. UDP), which are what NetMeeting uses to transmit video and audio. Alternatively, NetMeeting's TCP connections are assigned a slightly lower precedence value (16) to ensure that signalling messages between NetMeeting clients get through as well during periods of very high network congestion.

Figure 6 shows a different policy, which says that a non-default level of service should be given to TCP connections whose destination is a particular machine, one whose IP address is 192.168.10.10. We implement this by intercepting the `Connect` method, rather than `Socket` in the previous example, since it is with `Connect` that we discover where the connection is headed.

While this example is of a host, it should be clear how per-subnet policies can be implemented by means of, for instance, Tcl's regular expression facilties.

```
#
# Give different service to a particular destination
#

proc Connect { pid uid cmd fd address sqos gqos } {
   global AF

   # Parse the generic address argument
   set family [lindex $address 0]
   if { $family == $AF(INET) } {

      # Parse an IPv4 destination address
      set inaddr [lindex $address 1 ]

      if { $inaddr == "199.2.53.102" } {
         return "TOS 48"
      }
   }
}
```

Figure 6: Example script to deliver better service a particular destination host

4.3 Network Functionality

In order for the marking of IP packets to have any meaning, of course, it is essential that the network elements which queue and forward packets can detect and interpret the packet markings in a useful way, and apply different queueing and/or forwarding behaviour to them. Such an idea is not new, indeed it is the basis of the definition of the IP header Type Of Service octet, which dates back to the early definitions of the Internet Protocol. However, not all types routers in service in the commercial Internet are capable of supporting the ToS byte, and interpretations often vary within the definition. The most common implementation in widespread commercial deployment is Cisco's Committed Access Rate facility.

Just as important, however, to the effective deployment of differentiated forwarding policies in the network is the need for implementations to be efficient, in particular to run at forwarding rates as high as when the facility is not enabled. Most current generation routers which offer some kind of type-of-service or diffserv functionality do not implement it as part of the forwarding fast path. Instead, packet forwarding when type-of-service detection is enabled is carried out in software by a processor on the router, bringing the capacity of the router down to levels similar to the BSD or Linux-based routers used for much networking research. Such routers offer considerable flexibility in protocol processing, but do so at the cost of delivering performance unsuitable for a large commercial network. Consequently, there is an overwhelming disincentive for network operators to enable the functionality in the network equipment.

Fortunately, most next-generation edge routers are capable of making forwarding decisions on the fast path (in hardware) based on much more of the packet header, including the type-of-service octet, and also of queueing packets destined for the same output port differently. In our demonstration we used a PacketEngines PowerRail 5200 Enterprise Routing Switch [Pac98], which will route IP packets at line rates (100Mbit/second Fast Ethernet is this case), queue packets on output ports differently according to the priority field of the Type-Of-Service octet. Such functionality is now relatively common in new IP routers. The PacketEngines switch does not provide true DiffServ per-hop behaviours (PHBs), currently being defined by the IETF DiffServ working group, but such facilities are now appearing.

5 DEMONSTRATION

We have built a demonstration setup to illustrate the operation of the system. The configuration is shown in figure 7.

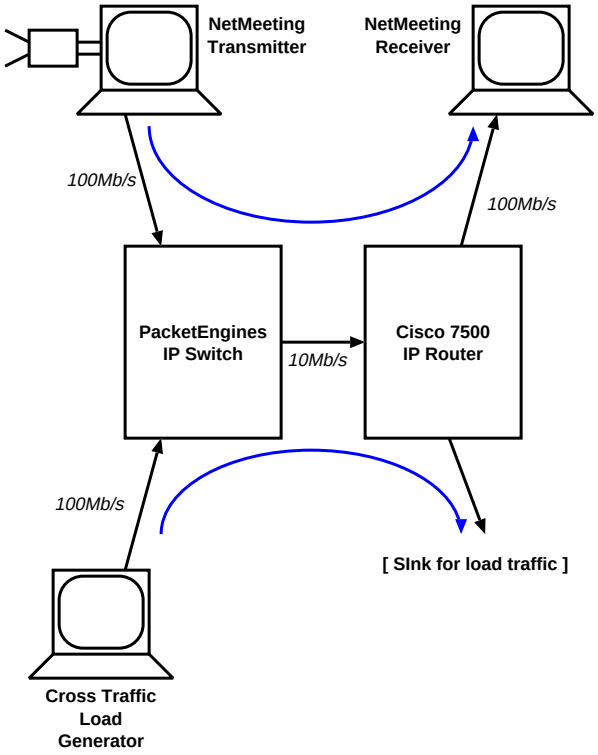

Figure 7: Demonstration network configuration

At the centre of the demonstration is a PacketEngines switch configured to queue packets on its output ports differently according to the precedence field of the type-of-service octet in the IP header. The switch will direct packets to one of eight output queues per port, based on the 3-bit field, and will then service those output queues using a Weighted Fair Queueing (WFQ) algorithm with the highest weight corresponding to the highest ToS precedence.

For our purposes the only important details here are that packets are given different service levels according to the ToS octet—any router which can forward in this way would work for us here. However, use of the WFQ algorithm does have significant advantages over a simple priority scheme: even under very high network load and congestion, all classes of traffic will receive some service under WFQ, something which is not the case for a priority scheme.

Two PC running Windows NT 4.0 are connected to the switch/router by 100 Mb/s Fast Ethernet links. The first has the QoS agent installed on it, and runs an unmodified version of NetMeeting which can transmit video streams (and audio streams) across the network to a receiving machine. We configure the policy daemon on this machine with the example script shown in figure 5, which gives assigns a higher importance to NetMeeting control connections (using TCP), and the highest importance to NetMeeting video and audio connections (which use UDP).

The second PC acts as a cross-traffic generator. It runs a simple program which can send back-to-back UDP packets to any destination IP address, and is capable of completely saturating a 100BaseT link. The program sends two seconds of load, then sleeps for two seconds sending no traffic, before starting again.

The outgoing link from the switch/router is an Ethernet link configured to run a 10Mb/s, and is connected to a router (in this case a Cisco 7500). The router in turn has two further interfaces configured. Attached to one of these is a third PC which also runs NetMeeting, to receive streams from the first PC. The other router interface has no attached hosts, but acts as a sink interface for packets from the load generator machine.

The configuration as a whole simulates the effect of sending video traffic (from the NetMeeting source to the NetMeeting sink machine) across a wide-area congested link (the 10Mb/s connection between the routers). The aim is to illustrate how, with appropriate configuration in network switches and marking of packets according to desired levels of service, performance of delay- and loss-sensitive applications, such as video conferencing with NetMeeting, can be dramatically improved without any modification to existing operating systems or applications.

The demonstration does not of course address the related issues, just as necessary in a real-world environment, of Service-Level Agreements (SLAs) between network providers and users, and the policing of such agreements at the edge of a providers network. These are crucial elements of the DiffServ framework.

5.1 Observed Behaviour

We do not present here any quantitative results of using the demonstration configuration. Exact measurements of here are not really the point, particularly as our experimental configuration is intended to be illustrative, rather than a basis for measurements that reflect real network conditions. Furthermore, the application behaviour we observe is sufficiently clear-cut that presenting quantitative measurements would not convey much useful information at this stage. Therefore, we simply present qualitative observations on our experiments.

We run the load generator to provide cross traffic and set up a NetMeeting video connection from the source PC to the sink PC. Without the QoS policy daemon running, the WinSock protocol stack in the source PC defaults to conventional behaviour with no outgoing network packets marked (a Type-of-Service value of 0). In this case, we see a complete breakup of the video connection for the 2-second periods when the load generator is sending cross traffic, even though the bandwidth required by the H.323 video connection (a few hundred kilobits at most) is more than an order of magnitude less than the bandwidth of the 10 Mb/s congesting link between the routers.

Such a result is unsurprising, but it is indicative of effects that occur in the Internet during transient periods of high network congestion.

We then start the QoS policy daemon on the source machine, and restart the NetMeeting application. This time, the protocol agent in the WinSock stack succeeds in contacting the daemon, and the policy embodied in the script causes the sockets created by NetMeeting to be configured to send packets with different Type-of-Service values. The first result we see is that the conference between the two NetMeeting applications is established much faster, since the TCP packets for the control connections receive much higher priority treatment. Secondly, and more importantly for our demonstration, the video quality is now unaffected by the cross traffic on the congested link, in contrast to before. We see no visible degradation in video quality whatsoever.

6 SECURITY CONSIDERATIONS

A number of concerns arise to do with security in the context of this work. They fall into two categories in terms of their consequences for users of the system and of the wider network:

1. Can users acquire an unfair share of network resources by abusing the mechanisms our system puts in place?
2. Can legitimate users be more exposed to denial-of-service attacks (whether service from the network, or service from the machine they use) by using this system?

The first issue is relatively easily addressed: our system adds no functionality to an individual application that could not have been added by the application programmers themselves. Setting the DiffServ octet for a socket is not a privileged operation on Windows NT (or Unix), and so while we may be making it easier to do this, we are not violating any existing privilege boundaries in this regard. The protocol agent runs as an unprivileged library in the same protection domain as the application itself.

More generally, of course, the problem of users "stealing" network resources by illegitimately marking packets is addressed in the DiffServ framework [BBC+98]: IP flows are individually policed at the edges of networks to ensure that they conform to service level agreements. Thus any unfairness in network usage (and consequently denial of service with respect to network resources) will be confined to within the organisation which is party to the service level agreement. Our system does not alter this.

The question of denial of machine resources is more difficult. The protocol agent can only cause damage if it is connects to a rogue policy daemon which tells it to incorrectly mark packets; written robustly, TCP timeouts and careful checking of the return value (for buffer overruns, etc.) should prevent outright failures. It's impossible to connect to the daemon from across a network, but there is no authentication between the daemon and protocol agents (other than the RPC service identifier).

It is possible that a malicious program running on the same machine could contact the protocol daemon and send bad strings to it. SafeTcl here prevents the strings from having any effects external to the interpreter itself, also it should be noted that since the protocol daemon only performs calculation on strings and returns strings, it does not need to have any system privileges at all. The major threat here is denial-of-service: while we can guarantee that protocol agents can only send well-formed and easily executable strings at the daemon, a malicious program sending it an infinite loop would tie it up indefinitely.

In practice, the NT servers and workstations where one might expect to run this system are effectively single-user machines, and so the dangers of malicious programs being introduced are no different with our system than on machines without it. Since the interactions any component has with the network are limited to packet marking, the system is no more or less secure than vanilla Windows NT.

7 CONCLUSIONS

We have a described a system for assigning network Quality of Service characteristics to network flows originating with preexisting applications written without reference to QoS issues. Two related techniques are employed to achieve a practical solution to the problem of support for legacy applica-

tions.

Firstly, the system uses a "protocol agent" inserted into the Winsock protocol stack, so as to avoid the need to implement specific mechanisms in either applications (not an option for legacy programs) or the operating system itself (we use unmodified Windows NT). This protocol agent provides the mechanism required to intercept networking calls and mark the packets.

Secondly, we employ a domain-specific language technique in the implementation of a daemon to encapsulate the policy decisions as to how to mark packets of particular flows. This provides a per-machine point of control for allocation of resources, which can be naturally extended to a distributed management framework.

Additionally, we make use of the capabilities of newly introduced network switches and routers to provide differentiated treatment of network flows based on packet marking.

Finally, we describe an experimental configuration which clearly demonstrates the benefits of the system on the performance of a sample multimedia application, NetMeeting.

The guiding principle of this work has been to enable a transition from a situation of best-effort network facilities in the Internet, and multimedia applications written based on this assumption, to an environment where the network is capable of delivering multiple levels of service, and applications are written to take advantage of these facilities. By intervening at a single point—taking advantage of new network facilities *without* requiring new applications to be written and deployed—we hope to aid in the bootstrapping process.

8 ACKNOWLEGEMENTS

Thanks are due to Bryan Lyles, for guidance, moral support and pizza during the long evening hacking the switch.

REFERENCES

Alm92 P. Almquist. Type of Service in the Internet Protocol Suite. Internet Request for Comments no. 1349, July 1992.

BBBN98 F. Baker, D. Black, S. Blake, and K. Nichols. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. Internet Request for Comments no. 2474, December 1998.

BBC+98 D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. Internet Request for Comments no. 2475, December 1998.

BBH+97 S. Berson, R. Braden, S. Herzog, S. Jamin, and L. Zhang. Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification. Internet Request for Comments no. 2205, September 1997.

BSY+97 Yoram Bernet, Jim Stewart, Raj Yavatkar, Dave Andersen, and Charlie Tai. Winsock2 Generic QOS Mapping. Microsoft Developer Network Library, 1997. version 2.6.

CCG+93 Andrew Campbell, Geoff Coulson, Francisco Garcia, David Hutchinson, and Helmut Leopold. Integrated quality of service for multimedia communication. In *Proc. IEEE INFOCOMM'93, San Francisco*, March 1993.

HOB99 Wei Hua, Jim Ohlund, and Barry Butterklee. Unraveling the Mysteries of Writing a Winsock 2 Layered Service Provider. *Microsoft Systems Journal*, May 1999.

LMB+96 I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, and R. Fairbairns. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

MP96 D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. Second Usenix Symposium on Operating System Design and Implementation*, pages 153–168, 1996.

Nic90 C. Nicolaou. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, April 1990.

OLW97 John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl Security Model. Sun Microsystems Laboratories Technical Report no. TR-97-60, March 1997.

Pac98 PacketEngines, Inc. The Wire-Speed Routing Guide. `http://www.packetengines.com/education/techpapers/routing/`, May 1998.

Wel97 Brent B. Welch. *Practical Programming in Tcl & Tk*. Prentice Hall, 2nd edition, July 1997.