Proceedings of the 7th USENIX Tcl/Tk Conference

Austin, Texas, USA, February 14–18, 2000

INTRODUCING QOS AWARENESS IN TCL PROGRAMMING: QTCL

Roberto Canonico, Maurizio D'Arienzo,
Simon Pietro Romano, and Giorgio Ventre

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

© 2000 by The USENIX Association. All Rights Reserved. For more information about the USENIX Association: Phone: 1 510 528 8649; FAX: 1 510 548 5738; Email: office@usenix.org; WWW: http://www.usenix.org. Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes.This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Introducing QoS awareness in Tcl programming: QTcl

Roberto Canonico, Maurizio D’Arienzo, Simon Pietro Romano, and Giorgio Ventre
Dipartimento di Informatica e Sistemistica, Università di Napoli “Federico II”, Napoli, Italy
{canonico, darienzo, sprom}@grid.unina.it ventre@unina.it

Abstract

A number of distributed applications require communication services with Quality of Service (QoS) guarantees. Among the actions undertaken by the Internet Engineering Task Force (IETF) with regard to the end-to-end QoS provisioning issue in the Internet, the definition of the Integrated Services (*IntServ*) framework plays a major role. According to this model, applications need to interact with network routers by means of a signalling protocol, RSVP. Even though special-purpose APIs have been defined to let applications negotiate QoS parameters across RSVP-capable networks, the integration of QoS negotiation mechanisms in the applications still remains an open issue. In modern programming, the Tcl scripting language plays a significant role, as it enables fast system prototyping by gluing basic components to build complex applications. In this paper we present QTcl, an extension of Tcl-DP which provides programmers with a new set of primitives, fully compliant with the SCRAPI programming interface for RSVP. We also present how QTcl has been used in an advanced VoD application to setup reservations in an IntServ network.

1. Introduction

Building global-scale distributed systems with predictable properties is one of the great challenges for computer systems engineering in the new century. Quality of Service (QoS) requirements will be critical for a large number of these systems, in particular for distributed applications whose performance depends mainly on the characteristics of the communication service provided by the networking infrastructure [1]. The global network par excellence is the Internet, with millions of users spread world-wide. Communication on the Internet is based on the connectionless IP protocol, which offers only a best-effort service. Hence, a great effort has been made in the past years to provide advanced communication services with QoS guarantees in IP-based networks.

The Internet Engineering Task Force (IETF) has defined an Integrated Services (*IntServ*) [2] framework to provide a service model that includes best-effort service, real-time service and controlled link sharing. According to this model, applications can, with the help of an appropriate signalling protocol like *RSVP (Resource reSerVation Protocol*) [3], request communication services with bounds on communication throughput or end-to-end latency. To do so, network routers need to implement special resource management policies and packet scheduling algorithms. However, to build distributed systems which benefit from the advantages of new networking services, we need to design QoS-aware applications, i.e. applications that know exactly their communication requirements and are able to interact with the network to negotiate the quality of the communication service. Hence, the need of defining ad-hoc APIs, which let applications issue per-stream resource reservations.

A large number of applications, in particular multimedia applications, consist of a set of pre-existing components (*building blocks*) glued together by a common GUI. To develop applications of this kind, scripting languages have proved to be better suited than system programming languages [4]. It is then reasonable to provide support for QoS into modern scripting languages. In this paper we present QTcl, an extension of Cornell’s Tcl-DP. QTcl extends the Tcl-DP interpreter by providing a set of new commands, according to the SCRAPI application programming interface, defined by the IETF [5]. This API conforms to a simplified model, in order to reduce the complexity of the development of new QoS-aware applications.

The rest of the paper is organised as follows. In section 2 we briefly describe the *IntServ* model and the SCRAPI programming interface. In section 3 we present QTcl and the set of new commands. We show the use of QTcl commands in a simple application in section 4. In section 5 we illustrate how we have implemented QTcl, as an extension to Cornell’s Tcl-DP. Finally, in section 6 we present a distributed VoD application, which uses QTcl to protect its data flows in a QoS-enabled internetwork.

2. QoS in the Internet

IP has been playing for several years the most important role in global internetworking. Its connectionless nature has proved to be one of the keys of its success.

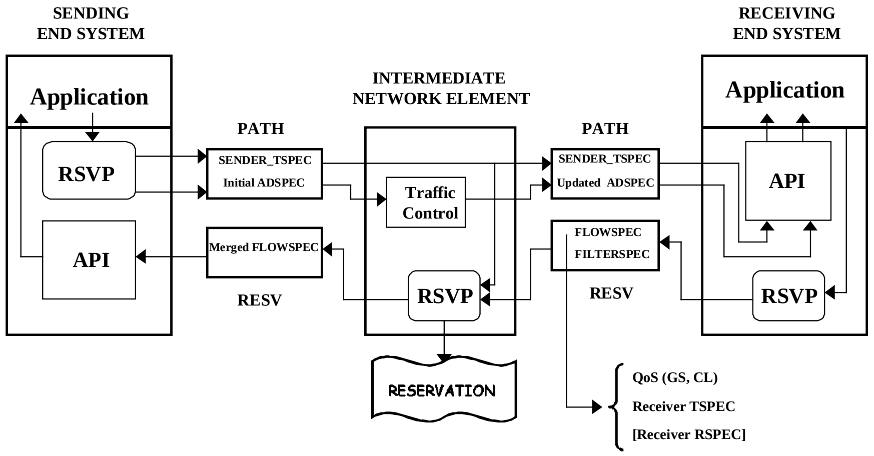

Figure 1: The use of RSVP objects during the resource reservation phase

Based on this assumption, the IETF Integrated Services working group has specified a control QoS framework [2] in order to provide new applications with the appropriate support. Such a framework proposes an extension to the Internet architecture and protocols which aims at making broadly available integrated services across the Internet.

The key assumption on which the reference model for integrated services is built is that network resources (first of all its bandwidth) must be explicitly managed in order to meet application requirements. The overall goal in a real-time service, in fact, is that of satisfying a given set of application-specific requirements, and it seems clear that guarantees are hardly achieved without reservations. Thus, resource reservation and admission control will be playing an extremely important role in the global framework. The new element that arises in this context, with respect to the old (non-real-time) Internet model, is the need to maintain flow-specific state in the routers, which must now be capable to take an active part in the reservation process.

Based on these considerations, the components included in the reference framework are a *packet scheduler*, an *admission control module*, a *packet classifier* and an appropriate *reservation setup protocol*. The first three modules together form the *traffic control interface* of the router. The reservation setup protocol is needed to create and manage state information along the whole path that a specific flow crosses between two network end-points. One of the features required to such a protocol is that of carrying the so-called *FLOWSPEC* object, that is a list of parameters specifying the desired QoS needed by an application. At each intermediate network element along a specified path, this object is passed to admission control to test for acceptability and, in the case that the request may be satisfied, used to appropriately parameterize the packet scheduler [6]. RSVP [3] is the resource reservation protocol recommended by *IntServ*.

Data treated by RSVP are of three natures, according to the entity that supplies them (sender and receiver) or modifies them (intermediate network elements). The information supplied by each sender, and conveyed in the *SENDER_TSPEC* object, concerns the type of traffic that it is going to generate. Receivers provide *FLOWSPEC* objects, which are built of two parts, *RECEIVER_TSPEC* and *RECEIVER_RSPEC* (both contained into a message called **RESV**). The former contains the traffic description the resource reservation should apply to, while the latter carries the service class to be used and the corresponding quality of service parameters.

Intermediate network elements, in turn, provide additional information such as available services, delay and bandwidth estimates, and additional service specific parameters. This information is contained in *ADSPEC* objects, and is used by the receivers to choose a service and determine the reservation parameters. *ADSPEC* and *SENDER_TSPEC* objects are both contained into a **PATH** message. Figure 1 shows the use of the defined messages during the resource reservation phase.

IntServ service classes define a framework for specifying services provided by network elements and available to applications, in an internetwork capable of offering multiple, dynamically selectable qualities of service. So far, two different service classes have been defined: *Guaranteed Service* (GS) [7] and *Controlled Load* (CL) [8].

In both cases, it is required that the sender provides, in *TSPEC* objects, a description of the traffic it is going to generate. Since traffic patterns are complex to describe, a worst case characterisation is provided (*traffic envelope*), according to a token bucket model [9]. Relevant parameters are the following:

- token bucket depth (b [Bytes]),
- average rate (r [Bytes/s]),
- peak rate (p [Bytes/s]),
- minimum policed unit (m [Bytes]),
- maximum datagram size (M [Bytes]).

A source conforming to such a description will generate, during any time interval of length τ, a number of bytes upper bounded by (Fig. 2):

$$A(\tau) = \min(b + r\,\tau,\; M + p\,\tau), \qquad \tau \geq 0$$

Guaranteed Service (GS) provides the clients data flow with firm bounds on the end-to-end delay experienced by a packet while traversing the network. It guarantees both bandwidth and delay. The GS emulates the service that would be offered by a dedicated communication channel between the sender and the receiver. Two parameters apply to this service: *TSPEC* and *RSPEC*. The *TSPEC* describes the traffic characteristics for which service is being requested. The *RSPEC* specifies the QoS a given flow demands from a network element. It takes the form of a *clearing rate* R and a *slack term* S. The clearing rate is computed to give a guaranteed end-to-end delay and the slack term denotes the difference between desired and guaranteed end-to-end delay after the receiver has chosen a value for R.

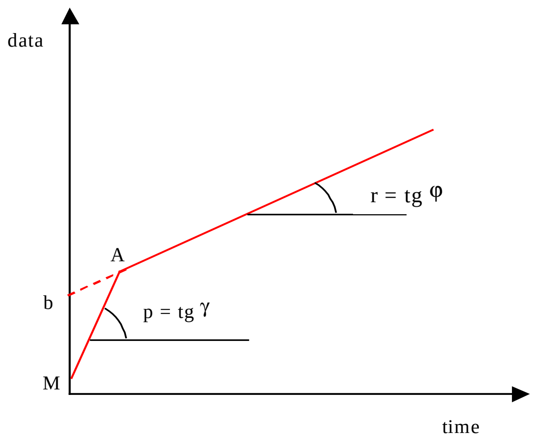

Figure 2: Traffic envelope for a source conforming to a (b, r, p) token bucket

Controlled Load (CL) service, on the other hand, may be thought of as a "controlled best-effort" service, i.e. a service with the same characteristics of a best-effort delivery over a not overloaded network. To avoid QoS degradation when the network load increases, CL relies upon admission control algorithms. CL is best suited to applications that have been developed taking into account the limitations of today's Internet, but are highly susceptible to overloaded conditions. A typical example is given by adaptive real-time applications, which have proved to work well when there is little or no load on the network.

2.1. QoS programming interfaces

In the framework we just depicted, the IETF has defined RAPI [10], an API compliant with the RSVP Functional Specification [3]. It is a user-level library written in C, which can be used by applications aiming at exploiting the QoS functionalities made available by a network reservation protocol like RSVP. RAPI calls let an application interact with a local RSVP daemon process, in order to establish a communication with QoS guarantees.

The RAPI interface is a first step towards the integration of communication services with QoS guarantees into applications; yet, its use is somewhat complex, since the application programmer must be aware of a number of parameters concerning the reservation. To cope with such problems, the IETF has proposed a simpler programming interface, layered on top of the RAPI and called SCRAPI [5]. SCRAPI provides only three functions:

- *Scrapi_sender*, to be used by the sender of a data stream associated to an RSVP session,
- *Scrapi_receiver*, to be used by the receiver, and
- *Scrapi_close*, to close an RSVP session.

Figure 3 shows the SCRAPI state diagram. A generic host starts in the **Closed** state. Then, it can act either as a sender or as a receiver or as a sender/receiver.

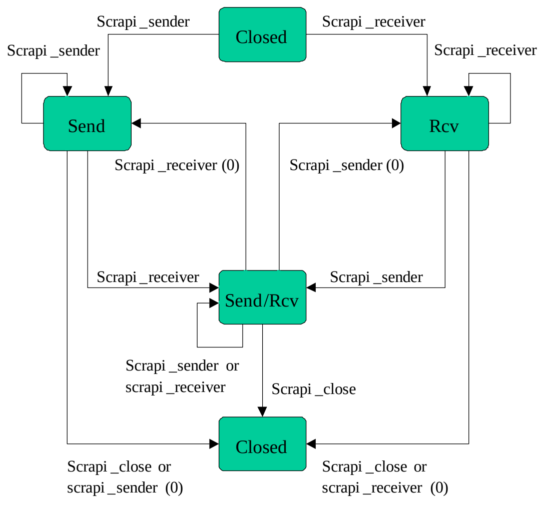

Figure 3: State diagram for the SCRAPI inter-

SCRAPI differs from RAPI especially in the error-handling model. While RAPI requires the application programmer to implement a set of upcall routines, to handle asynchronous events and errors, this is not required anymore when using SCRAPI. Upcalls are replaced by a simplified "three colours" error model, which makes use of three different values (red, yellow, green), whose combinations let the application know the state of a reservation at a given instant in time. A reservation is said to be in the RED status if the transmission of **PATH** messages from the sender has not started yet, or **PATH** messages have not arrived at the respective receiver yet, or the system is currently in an error state. A YELLOW state indicates that a valid **PATH** message flow is present, but reservations have not been made by receivers. The transition to the GREEN state happens when the reservation is accepted. This strategy leads to a "light-weight" model, even if it imposes a number of constraints on the QoS negotiation process.

The service model used by the simplified interface builds the required RSVP objects in a way that is transparent to the end user. In particular, the object *SENDER_TSPEC* T:

[*token_bucket_rate* *token_bucket_depth*
min_policed_unit *max_datagram_size*]

is created based on the following assumptions:

- the average bandwidth, specified by the sender, is actually used as the token bucket rate (r) for the flow;
- the peak rate (p) is considered infinite;
- the token bucket depth (b) is assumed twice as the average bandwidth (and so two times r);
- the minimum policed unit (m) is 64 bytes;
- the maximum policed (M) unit is the greatest MTU associated to the IP interfaces available on the host.

As far as the receiver is concerned, the RSPEC object for the Guaranteed Service is built by simply setting the value of the clearing rate (R) to the average bandwidth and using a slack term (S) of zero. In the case of Controlled Load service, the receiver reserves as much bandwidth as the sender declares in the *SENDER_TSPEC* (r).

3. QTcl API

The SCRAPI programming interface has already been implemented as a C library, and used in modified Mbone tools. A support for QoS communication in Tcl applications, instead, was not available. Since we wanted to implement in Tcl a QoS-aware application for the distribution of multimedia documents, we have developed QTcl, an extension of the Tcl scripting language which implements the SCRAPI interface. QTcl provides the Tcl programmer with a set of new commands to create reservations in an RSVP-enabled internetwork. The new commands are shown below:

```
• dp_scrapiSender   dest_hostname
                    dest_port
                    source_hostname
                    source_port
                    bandwidth
                    protocol
• dp_scrapiReceiver dest_hostname
                    dest_port
                    source_hostname
                    source_port
                    service
                    protocol
• dp_scrapiStatus   dest_hostname
                    dest_port
                    protocol
• dp_scrapiClose    dest_hostname
                    dest_port
                    source_hostname
                    source_port
```

Using these commands, it is possible to manage the whole process of reservation setup.

The bandwidth parameter must be expressed in Bytes/sec. The `service` parameter can be one of the following two values: `cl` indicating Controlled Load or `gs` indicating Guaranteed Service. Finally, the `protocol` parameter can be either `tcp` or `udp`.

`dp_scrapiSender` opens an RSVP session and starts **PATH** message transmission from source host to destination host. **PATH** messages are refreshed every 30 seconds.

`dp_scrapiReceiver` is invoked by a receiver in order to make a reservation request. The receiver specifies the desired QoS and class of service (Guaranteed Service or Controlled Load) according to the information contained into the **PATH** message. This request is forwarded to the sender across the network via a **RESV** message. After sending a **RESV**, the receiver waits for a confirmation of successful reservation from the sender for at most 10 seconds, as set by a specific timer; however, even in case of timer expiration the reservation process will go on.

`dp_scrapiStatus` allows to verify the current status of a session, according to the simplified error model available in the SCRAPI interface, i.e. it returns a RED, YELLOW or GREEN value according to the status of the RSVP session.

`dp_scrapiClose` is the function called to tear down an RSVP session, both in reception and in transmission.

4. A simple QTcl application

Figure 4 shows a simple application made of a sender process and a receiver process. The two processes should be executed on different hosts connected by an RSVP-enabled internetwork. The sender process invokes the `dp_scrapiSender` command, to start the transmission of **PATH** messages and then waits in a loop until the reservation is completed. The receiver process, instead, issues the `dp_scrapiReceiver` command to start the transmission of **RESV** messages and waits for the reservation to be completed. As soon as the reservation is completed, the sender starts transmitting UDP messages, 1480 bytes in length. The receiver, in turn, measures the time needed to receive a number N of such messages and estimates the received throughput. This simple application can be tested in order to verify that the achieved throughput is independent from the network conditions, as long as the routers implement an *IntServ* Guaranteed Service.

5. QTcl implementation

QTcl has been conceived as a tool for supporting the development of distributed applications with simple QoS requirements. As we did not want to reinvent the wheel, we felt that some useful features were already available in the Tcl-DP extension, developed at Cornell University [11]. In particular, we found the `dp_RPC` mechanism particularly suitable to support the receiver-initiated reservation mechanism of RSVP. Hence, QTcl has been developed starting from the original Tcl-DP source distribution. We then extended the Tcl interpreter by creating a set of C functions that implement the SCRAPI primitives.

Notice that SCRAPI is only a programming interface to access the RSVP service, which must be implemented by a proper operating system module. In UNIX-like systems, this is usually a daemon process, which runs with root privileges in the end systems. Our current implementation of QTcl is available for the SUN Solaris, FreeBSD and Linux operating systems. For these systems, an RSVP implementation is provided by ISI [12].

Sender.tcl

```
# Sender
#!/home/qtcl/bin/tclsh8.0

package require dp

set sender [dp_connect udp -host
143.225.229.105\
   -port 3000 -myaddr localhost -myport 5000]

dp_scrapiSender 143.225.229.105 3000 \
   143.225.229.116 5000 100000 udp

while {$status != "green"} {
 after 1000
 set status [dp_scrapiStatus 143.225.229.105\
                              3000 udp]
}
puts $status

set pkt ""
for {set i 0} {$i < 1480} {incr i} {
 append pkt x
}

puts "Press ctrl-C to interrupt ...."

while {1} {
 set lun [dp_send $sender $pkt]
}

close $sender
```

Receiver.tcl

```
# Receiver
#!/home/qtcl/bin/tclsh8.0

proc bench { N } {
  global receiver
  set count 0
  while {$count < $N} {
    set rcv [dp_recv $receiver]
    incr count [string length $rcv]
  }
}

package require dp

set receiver [dp_connect udp -myport 3000]
fconfigure $receiver -blocking 1

dp_scrapiReceiver 143.225.229.105 3000 \
              143.225.229.116 5000 gs udp

while {$status != "green"} {
 after 1000
 set status [dp_scrapiStatus 143.225.229.105\
                              3000 udp]
}
puts $status

set N  10485760
set T  [lindex [time { bench $N }] 0]
set BW [format "%2.3f" [expr $N*8.0/$T]]

puts "Elapsed time: $T microseconds"
puts "Estimated bandwidth: $BW Megabit/sec"

close $receiver
```

Figure 4: A simple client-server QTcl application

As for the Microsoft Windows operating systems, the implementation of QTcl is not straightforward, due to the different semantic of the Microsoft RSVP-API implemented as part of their Winsock2 API. However, we are currently investigating the possibility of undertaking the port of QTcl for this platform.

Figure 5 shows the global picture of a UNIX host running a QTcl application and interacting with an RSVP-enabled router.

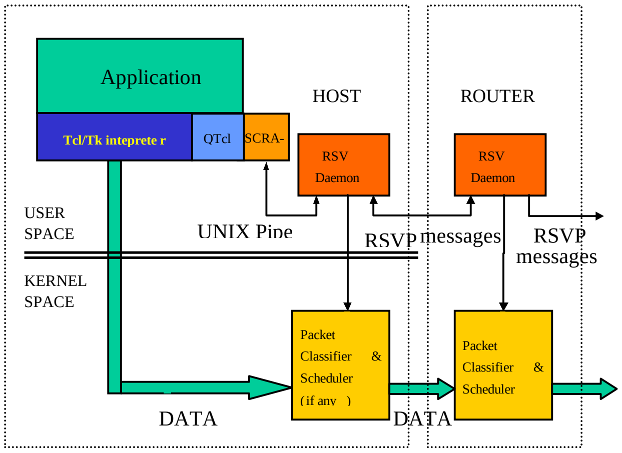

Figure 5: QTcl implementation in a UNIX host

5.1. Why Tcl-DP ?

Our latest version of QTcl has been developed from the release 4.0 of Tcl-DP. Tcl-DP communication services rely on different transport mechanisms: serial links, TCP, UDP, IP-multicast, and e-mail. Tcl-DP 4.0 is implemented as a loadable module, i.e. Tcl-DP commands are made available to the Tcl interpreter by means of the `package` command. Table 1 shows some of the Tcl-DP primitives that can be used to build a QoS-aware distributed application.

`dp_RDO`	Perform a remote procedure call without return value
`dp_RPC`	Perform a remote procedure call
`dp_MakeRPCServer`	Create a TCP RPC server channel
`dp_MakeRPCClient`	Create a TCP RPC client channel

Table 1: Tcl-DP commands

The RSVP protocol uses a receiver initiated approach. The practical consequences of this approach are different whether the application is based on a multicast or unicast communication. In a unicast based application (e.g. a Video on Demand system), a sender does not know in advance the address of the receiver. Hence, it can start sending **PATH** messages only after the receiver has declared explicitly its will of starting a session with resource reservations. To write such an application, the RPC mechanism provided by the Tcl-DP extension is extremely useful. Figure 6 provides an example of a unicast-based client application which uses a combination of the `dp_RPC` and `dp_scrapiSender` primitives to setup a reservation. The `dp_RPC` primitive invokes on a remote host a Tcl procedure, `ScrapiSndRsv`, which, among other things, in turn invokes the `dp_scrapiSender` primitive.

```
# Tell Video source to start sending PATH
msgs
if [catch {dp_RPC $sockV -timeout 60000 \
       ScrapiSndRsv \
       $obj(viAddrSrc,$urlSP) \
       $obj(viAddrDst,$urlSP) \
       $bwVideo $service} error] {
   catch {diva_CloseRPC $sockV}
   error "Server not responding: $error"
}
# Start sending RESV msgs upstream
set status ""
dp_scrapiReceiver \
       [lindex $obj(viAddrDst,$urlSP) 0] \
       [lindex $obj(viAddrDst,$urlSP) 1] \
       [lindex $obj(viAddrSrc,$urlSP) 0] \
       [lindex $obj(viAddrSrc,$urlSP) 1] \
       $service udp
while {$status!="green"} {
   after 1000
   set status [dp_scrapiStatus \
       [lindex $obj(viAddrDst,$urlSP) 0] \
       [lindex $obj(viAddrDst,$urlSP) 1]
udp]
}
```

Figure 6: Use of QTcl in combination with `dp_rpc` in a multimedia application to setup an RSVP session for a unicast stream.

6. A QoS-aware distributed multimedia application based on QTcl

To show the effectiveness of RSVP bandwidth management in a real application, we added the ability of making network resource reservations to DiVA, a distributed multimedia application developed by our research group. DiVA is capable of playing and controlling remote audio/video documents in streaming mode. Figure 7 shows the relevant data streams produced by the DiVA application among a streaming server host

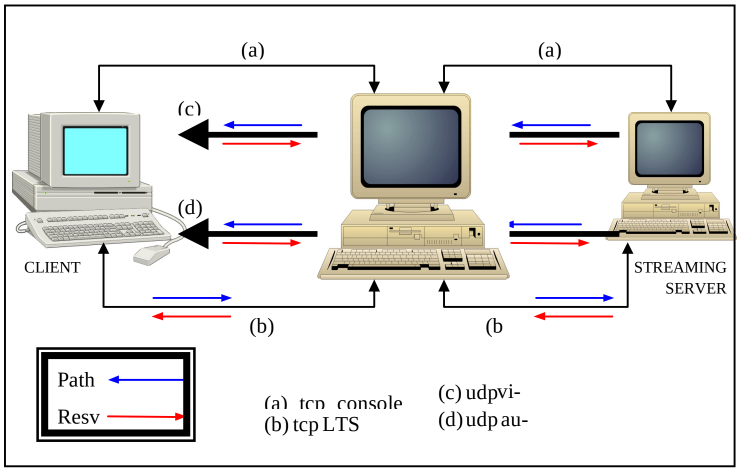

Figure 7: Data streams generated by the DiVA application and associated to RSVP sessions.

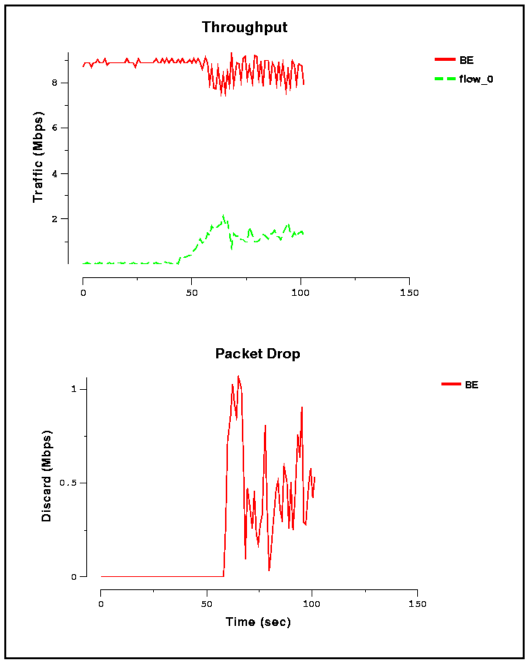

Figure 8: Streams behaviour with a reservation for the multimedia flows

and a client host. In particular, the UDP audio and video streams are transmitted downstream from the server on the right to the client on the left, while two TCP bi-directional streams are used to exchange control (console) and synchronization (LTS) information.

We tested the application in a testbed formed by two different Local Area Networks, connected by means of a WFQ router implemented in FreeBSD [6]. The router was connected to the first LAN through a 100 Mb/s Fast Ethernet card and to the second LAN through a 10 Mb/s Ethernet card. A host in the 10 Mb/s LAN acted as a client, while another host in the 100Mb/s LAN ran the DiVA video server. Hence, multimedia traffic flowed through the WFQ router.

To test the effectiveness of the traffic control mechanism implemented in the router, and the ability of the application to request the necessary Quality of Service, we generated a 9 Mb/s cross traffic stream among a pair of different hosts. Cross traffic and DiVA multimedia streams competed in the router for the 10 Mb/s bandwidth available in the Ethernet LAN.

In a first experiment, we did not make any reservation for the video/audio streams. In this case, multimedia traffic was not protected from the cross traffic and it had to share packet losses with it. Even though DiVA is capable of adapting the traffic generated to the available bandwidth, it was impossible to obtain a Quality of Service adequate for intelligible video and audio rendering in this case.

In a second experiment, we used RSVP to make a reservation for the flows generated by the video server, while cross traffic was still served as best effort.

In this case, Figure 8 shows that multimedia flows were fully protected from the best effort traffic, which started losing packets as soon as data streaming from the DiVA server began. This behaviour preserved a very good quality of video/audio rendering, in spite of the presence of cross traffic.

In our prototype, the bandwidth values used to setup reservations for the video and audio streams were determined empirically for each archived document, by observing the traffic produced by the application while streaming it. In a real-world application, these values should be retrieved by the client application in the form of *metadata* associated to the document.

7. Conclusions

An increasing number of distributed applications can benefit from the availability of improved communication services in RSVP-enabled IP internetworks, by acting in a proactive way, instead of passively adapting to the available QoS offered by current best-effort services. We believe that this support is helpful for a wide range of modern distributed applications. In this paper we have presented QTcl, a QoS control API which is compliant with the IETF SCRAPI interface, and has been designed as an extension of the Tcl scripting language. An implementation of QTcl for UNIX-derived operating systems is available on the web at: http://www.grid.unina.it/qtcl.

References

[1] K. Kavi, J.C. Browne, and A. Tripathi. "Computer Systems Research: The Pressure Is On". *Computer* , Jan. 1999, pp. 30-39.

[2] R. Braden, D.Clark, and S. Shenker. "Integrated Services in the Internet Architecture: an Overview". IETF *RFC 1633*, July 1994.

[3] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification". IETF *RFC 2205*, September 1997.

[4] J.K. Ousterhout. "Scripting: Higher-Level Programming for the 21st Century". *Computer*, March 1998, pp.23-30.

[5] B. Lindell. "SCRAPI - A Simple 'Bare Bones' API for RSVP". IETF Internet Draft draft-lindell-rsvp-scrapi-02.txt, Feb. 1999.

[6] R.D'Albenzio, S.P. Romano and G. Ventre. "An Engineering Approach to QoS Provisioning over the Internet". Lecture Notes in Computer Science no. 1629, Springer, May 1999, pp. 229-245.

[7] S. Shenker, C. Partridge, and R.Guérin. "Specification of Guaranteed Quality of Service". IETF *RFC2212*, September 1997.

[8] J. Wroklawsky. "Specification of the Controlled-Load Network Element Service". IETF *RFC 2211*, Sep. 1997.

[9] S. Keshav. "An Engineering Approach to Computer Networking". Addison-Wesley, 1997.

[10] R. Braden and D. Hoffman. "RAPI -- An RSVP Application Programming Interface - Version 5". IETF Internet Draft draft-ietf-rsvp-rapi-01.txt, Aug. 1998.

[11] M. Perham, B.C. Smith, T. Jánosi, and I.K. Lam. "Redesigning Tcl-DP". Procs. of the Fifth Annual Tcl/Tk Workshop, Boston, 1997.

[12] USC Information Sciences Institute (ISI), http://www.isi.edu/rsvp/release.html