WWW Applications for an Internet Integrated Service Architecture

T. V. Do, B. Kálmán, Cs. Király, Zs. Mihály, Zs. Molnár, Zs. Pándi

Department of Telecommunications
Technical University of Budapest
Fax: +36 1 4633263
Email: do@hit.bme.hu, s8659pan@ural2.hszk.bme.hu
Pázmány Péter sétány 1/D, Budapest, Hungary

Abstract

The IP protocol suite based Internet provides a number of useful telecommunication services, therefore every day the number of hosts connected to Internet grows at an unprecedented rate. However, the Internet itself does not guarantee any Quality of Service (QoS) for user flows, which prevents the provision of good quality for applications over wide area. This fact leads to the development of the Integrated Service Architecture (based on the use of the Resource Reservation Protocol-RSVP) and recently the Differentiated Service Architecture. At present a vast number of vendors provide RSVP capable routers to build an IP Integrated Service network, therefore the acceptance of the Internet Integrated Service concept strongly depends on applications which are capable to request QoS.

This paper presents a solution to incorporate the RSVP protocol into the architecture of existing WWW applications, which are the most popular ones on Internet today. The proposed architecture is flexible enough to be applied with any Web browser and the Web server as well. The first software version of the proposed architecture has been released.

Keywords: WWW, proxy, IntServ, QoS, RSVP

1. Introduction

The IP protocol suite based Internet provides a number of useful telecommunication services, therefore every day the number of hosts connected to Internet grows at an unprecedented rate. However, the Internet itself does not guarantee any Quality of Service (QoS) for user flows, which prevents the provision of good quality for applications over wide area. This fact leads to the development of the Integrated Service Architecture (based on the use of the Resource Reservation Protocol — RSVP) and recently the Differentiated Service Architecture [1,2]. The Integrated Service Architecture defines several service classes that, if supported by the routers traversed by a data flow, can provide the data flow with certain QoS commitments. The level of QoS provided by these enhanced QoS classes is programmable on a per-flow basis according to requests from the end applications. These requests can be passed to the routers by network management procedures or, more commonly, using a reservation protocol such as RSVP which was designed to enable the senders, receivers, and routers of communication sessions (either multicast or unicast) to communicate with each other in order to set up the necessary router state to support the services [1].

At present a vast number of vendors provide RSVP capable routers to build an IP Integrated Service network, therefore the acceptance of the Internet Integrated Service concept strongly depends on applications which are capable to request QoS.

This paper presents a solution to incorporate the RSVP protocol into the architecture of existing WWW applications. The motivation for choosing Web is that Web based applications (downloading file using the HTTP protocol, playing audio or video streams) belong to the category of the most widely used applications on the Internet today. The proposed architecture consists of a forwarding proxy, a QoS console and a QoS broker with an RSVP daemon. Moreover it is general enough to be used with different Web browsers and Web servers as well.

The paper is organised as follows. Section 2 describes a proposed system architecture. Section 3 provides further details on the QoS console. A forwarding proxy is discussed in Section 4. Section 5 describes the functionality of the QoS broker. Section 6 provides a conclusion of this paper.

2. System architecture

2.1 Elements of the architecture

When something new is integrated into an existing architecture, it is important to introduce the fewest possible modifications of the existing elements. Therefore an architecture for an integrated Web application is proposed in Figure 1. The system architecture for serving QoS requests consists of the following entities:

- Proxy: The architecture is based on the fundamental fact of the HTTP protocol specification [3], which enables one or more intermediaries to be present in the request/response chain. The most common form of an intermediary is a proxy. A proxy is a forwarding agent, receiving requests for a URL in its absolute form, rewriting all or part of the message, and forwarding the reformatted request toward the server identified by the URL. The proxy module translates requests coming from a Web browser, accepts and interprets user commands from the QoS console to communicate with an RSVP daemon about the QoS settings and to inform the QoS broker on the requirements of the user.

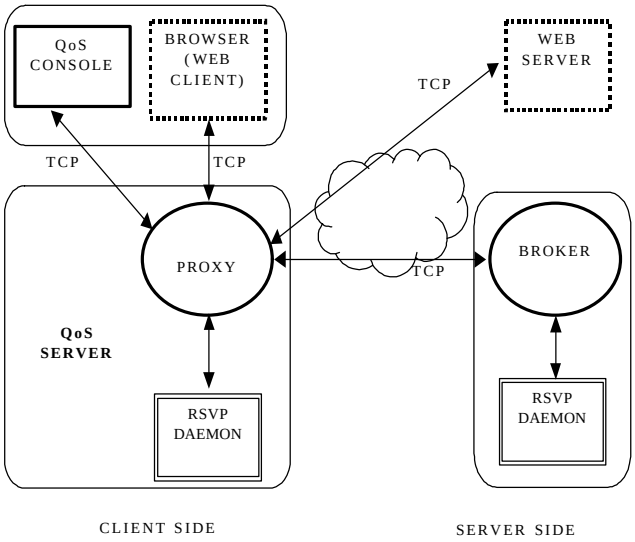

Figure 1

- The QoS console is a JAVA applet for controlling the settings of a flowspec on the client side. This module provides the graphical interface for the user to specify the flowspec (service class, Tspec and Rspec). The flowspec parameter will be transmitted to the proxy. Moreover, all the information of living TCP connections between the proxy and Web server will be displayed, therefore the QoS console also provides the opportunity to monitor and control connections between the proxy and the Web server.
- The RSVP daemon is responsible for directing the routers to satisfy the user's requirements with the use of RSVP. In this proposal we use the RSVP daemon implementation from the USC Information Science Institute [4].
- The QoS broker module communicates with the proxy about the user requirements and communicates with the RSVP daemon on the server side to direct it to start creating QoS sessions. This module runs on the same machine as the Web server program (although it can be placed in a host on the same LAN as the Web server is situated on). The proxy will open a TCP connection to the QoS broker (the port number is well-known for the QoS manager) and send all the necessary data. The responsibility of the QoS broker is to communicate with the RSVP daemon in order to generate the PATH message. The rationale behind the QoS broker is that the RSVP specification is receiver-oriented and it requires the sender to submit the PATH message [2].

Since QoS guarantee is often required between the ingress and egress router along a path, any host in the local LAN of the client can locate the proxy and any host in the local LAN of the Web server can locate the broker. If the browser and the proxy did not run in the same machine, a problem related to the identification of the user should be discussed and solved. In the first version of our implementation, this problem is not considered.

2.2 QoS request

The procedure for QoS Web requests is processed as follows:

1. The user configures his or her Web browser with the IP address of the proxy host and the port number of the proxy.
2. Then he or she starts a Web browser and downloads a JAVA applet (the JAVA applet will start a QoS console) which can be used for configuring flowspec settings.
3. He or she then starts browsing the Web.
4. Upon the HTTP request of the user agent, the browser sends an URL to the proxy. Then the proxy opens a TCP connection to the Web server. The proxy informs the QoS console about the connection and the user can assign a predefined flowspec for the TCP connection opened by the browser. The JAVA console sends the assigned flowspec to the proxy.

Then, the proxy sends the flowspec, the proxy IP address and port number, the server IP address and port number to the QoS broker through the new TCP connection between the proxy and the QoS broker. Next, the QoS manager communicates with the RSVP daemon to request QoS for the TCP connection between the proxy and the Web server.

5. If the downloading of the document ends, the Web server application will close the TCP connection. The user can also interrupt the downloading: the result is the same. The proxy clears the data about this connection from its database, informs the JAVA applet (to refresh its window), and also informs the client-side RSVP daemon. The RSVP daemon then tears down the path needed for the QoS request.

3. The QoS console (JAVA applet)

The QoS console provides a GUI in order to inform the user on what is happening in the background by making all the important events visible. Moreover, it provides controlling opportunity for the user. It should be absolutely user-friendly and foolproof. To make it more useful it should be platform independent, too. The JAVA language was an obvious selection for the implementation, because we can fulfill all the requirements by using JAVA.

The JAVA applet opens a TCP connection to the host it has just been downloaded from (thus no security violations in the browser should occur). After the mutual identification the console is going to be a passive "display" until the user interacts somehow (see Figure 2).

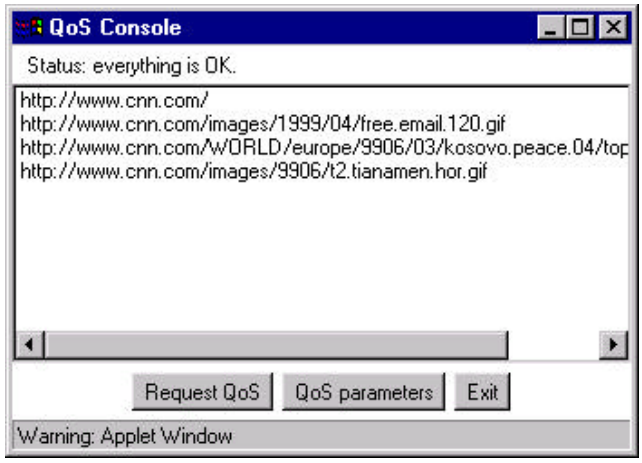

Figure 2

Its task is to maintain a list of the active TCP connections of the browser (members of this list are records consisting of an URL and a client-side port number) according to the data obtained from the proxy. It accepts two kinds of updating messages: one for signaling when a new connection is opened and one for signaling when an old one is closed. The console is always listening to the connection to the proxy and it displays the data about the connections until it stops.

The user may interact with the QoS console in two cases. He or she either modifies the current QoS parameters or makes a QoS request using them.

The first case is much simpler, because the reactions do not involve other parts of the proxy. In this case the console shows a parameter-window, where the settings can be altered (see Figure 3). These settings are stored until the console stops, even though the parameter-window is closed. In other words: there is always a current QoS parameter set, which can be displayed and altered by calling the parameter-window. To make the setting of the parameters easier the console offers parameter presets. If the parameter-window bothers the user, it can be hidden after the first use, but it is also capable of refining the parameters before each new request as more experienced users might need it.

Figure 3

In the second case the user makes a request. The console then becomes active; it forwards the current QoS parameter settings to the proxy together with the data about the selected connection. Its client-side port number identifies the connection. Then the console goes back to the passive state again. If a message arrives from the proxy concerning the result of the request, the console will notify the user about it. The console accepts two

more kinds of messages: one for signaling a successful request and one for describing the error occurred.

There are some other things to take care of. The console should handle the situation when the proxy closes the TCP connection between itself and the console. It should also recognize if it is the second instance running on the same host (the proxy will tell the console, if it finds a record in its database containing the same client host address). Last, but not least the console should be as robust as it can. The user and even the proxy should not make it run abnormally in any case.

4. Functional blocks of a Web QoS proxy

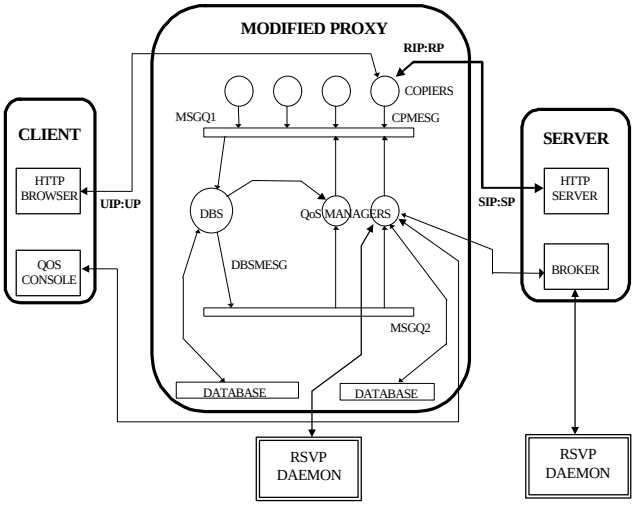

Figure 4

To simplify the explanation we assume that the proxy runs on the same machine as the browser, and the browser is properly configured for using our proxy. Thus the necessary communications between the proxy, the browser and the console will happen inside the client machine. Of course our proposal also allows the proxy to be situated on another machine (e.g. a local server).

In what follows we describe the working mechanism of the inner parts of the proxy in details. The main task of the proxy is to listen to a well-known port and to create a child process (called *copier* in the model) for each TCP/IP connection opened by the browser. After it is born, the *copier* sends a CPMESG to the message queue called MSGQ1. (This queue is maintained by the linux kernel, and is written only by the *copiers* and the *QoS managers*.) The process called *DBS* (our *DataBase Server*) reads the queue, and stores the content of the CPMESG into its own database. The proxy does nothing interesting (the TCP/IP packets pass through the *copier* process) until the user starts our *QoS console*, indicating that he or she wants to request some kind of QoS for an active TCP/IP connection.

The *QoS console* will establish a TCP/IP connection with the proxy (through a port number, which is considered to be well-known for the *QoS console*). This connection remains open until the *QoS console* closes it, or the proxy detects that the *QoS console* has died. The proxy handles this connection with another child process, called the *QoS manager*. After a *QoS manager* is born, it sends a QoSMMESG to MSGQ1 to ask the *DBS* to send the details of the active TCP/IP connections of the browser that runs on the same machine as the console. (This is necessary, however in the beginning the proxy will run on the same machine as the browser.) The *DBS* uses MSGQ2 to send data to the *QoS managers* about TCP/IP connections (DBSMESG). It also uses DBSSIG to tell the *QoS manager* that it should read MSGQ2 and test if it is still alive at the same time. As we can see, the identification of the users is done by their IP addresses: only one *QoS console* (on the client side) and one *QoS manager* process (in the proxy) runs for each IP address.

The task of the *QoS manager* is to represent the *console* inside the proxy: to inform the *QoS console* and to execute its commands.

The *QoS console* shows the list of the active TCP/IP connections to the user, and allows him or her to select an active connection (identified by the URL and the client-side port number, called User Port) and request QoS for it. The *console* sends the QoS details to the *QoS manager*, and waits for possible error messages. The *QoS manager* seeks in its database for the details of the appropriate connection, and then establishes a TCP/IP connection with the server-side *broker*. This small process does nothing but reads the parameters of the client-side QoS request through this connection, and makes the server-side RSVP-daemon start sending PATH_MSGs. Thus the sender-side part of the RSVP session is initialized, and the rest is done on the client side. This involves sending RSVP RESV_MSGs when the path is ready for the RSVP session, and the necessary initialization is done by the *QoS manager* via the RSVP daemon. Finally, if everything goes well, the selected TCP/IP connection gets the QoS, that was requested by the user.

The intercommunication messages used between the proxy and the other components of the system (the QoS console and the broker) are in ASCII format. All other messages that are used between inner components of the proxy are in binary format.

4.1 Copiers

The task of the copier is to manage HTTP requests of Web browsers, forwarding them to the respective Web servers. Thus the copier has access to all parameters needed for an RSVP session: i.e. to those of HTTP requests and of the TCP connection between the proxy and the Web server. The copier sends the required parameters to the DBS.

The copier consists of the following components:

- copier parent: it waits for TCP connections on the pre-defined port. When a Web browser initiates a connection, it creates a new copier child process for managing that connection. Then it waits for the next connection.
- copier child: each copier child manages one single TCP connection between a Web browser and the proxy. It communicates through three channels:
  - TCP connection from Web browser: for HTTP/1.0 persistent connections and for HTTP/1.1 several requests are allowed on a single TCP connection. Those requests can refer to several Web servers.
  - TCP connection to Web server: the copier opens this connection to the Web server requested. When a new request arrives for the same server, the TCP connection remains open. This is very useful for RSVP, as no new session has then to be initiated. If the request is for a new Web server, the old TCP connection is closed and a new one is opened.
  - Message queue towards the DBS (MSGQ1).

The copier child works as follows: when started, the copier gets the client side TCP connection from the parent, and waits for data. It processes the first line of the request that contains the URL including the name of the Web server. Then it tries to open a TCP connection to the Web server. If it fails, the browser is notified, and the copier child terminates. If successful, the URL and the parameters of the proxy — Web server connection are sent to the DBS in a CCREATED message. Then the copier sequentially processes the lines of the request header, and sends them with necessary modifications to the Web server. If the request contains a body as well, it is also sent.

The copier child then receives and processes the response, and forwards it to the Web browser. In case a HTTP/1.0 connection is non-persistent, it sends a CDIED message to the DBS, and immediately terminates. For HTTP/1.1 or a HTTP/1.0 persistent connection, copier child waits for further requests and/or responses. There are three possibilities:

1. A CURL message is sent to the DBS if a new response arrives.
2. A CDIED message is sent, when a request to a different Web server arrives. The TCP connection to the old server is closed, and the new one is opened. Then a CCREATED message is sent with the parameters of the new connection.
3. A CIDLE message is sent in any other case.

The copier child may terminate either because of time-out or when the browser closes the TCP connection.

4.2 DBS

The tasks of the DBS are administering the system components and managing the communication between them. It has to maintain a database about all the important data. Due to this, when a QoS manager is created, it will know about all the previously opened but still existing TCP connections of the user it is assigned to.

The DBS watches the first message queue (MSGQ1). When a valid message is received, the DBS processes it. In any other case the message is dropped.

When updating the database, the DBS first of all finds the record where it has to modify, insert or remove. The key is the user IP address in commands sent by the QoS manager, and user IP address and user port together in commands sent by the copier. If the record does not exist, or it already exists in case of an insert command, the message is semantically wrong. The DBS drops it and waits for the next message. The QCREATED (QoS manager created) is a special command. If a QoS manager already exists with the specified IP address, the DBS has to kill the new QoS manager immediately.

If the command was a TCP state command from a copier, the DBS has to check if there is a QoS manager interested in the changes. The modified record belongs to a user. If the user's main record has the field "QoS manager" filled, the DBS has to notify that QoS manager about the changes.

If the command was a QCREATED (QoS manager created) command, the DBS has to send all the URLs belonging to the user whose user IP address is specified in the command. The DBS formats the message and puts it into the second message queue (MSGQ2). Then it sends a signal to the QoS manager meaning that

there is a message waiting for it. The DBS can't send a message, while the destination QoS manager has unread messages in the queue, in order to avoid congestion. Semaphores help with solving the problem.

If the destination is died due to an internal error, the DBS has to remove the message from the queue and update the database, as in case of a QDIED (QoS manager died) command.

4.3 QoS managers

The QoS manager handles exactly one user's QoS requests. Its other role is to provide the user the list of active TCP connections opened by the browser via the proxy. For each console exactly one QoS manager is created.

Communication with the DBS

When a QoS manager starts, it informs the DBS about itself. From now on, the DBS will send information about the TCP connections concerning this QoS manager. If more then one QoS manager is invoked from the same IP address, the DBS will kill the superfluous ones. Those should exit without sending a death notification message to the DBS. The messages sent to the DBS are QCREATED, and QDIED. In the other direction, the DBS sends DNEWURL and DDELURL. These messages mean the start and the termination of a download process, respectively. The status of the TCP connections is traced using these messages.

Communication with the console

A new QoS manager is spawned, whenever a TCP connection is opened to the Proxy on a special port (which is "well known" for the QoS console). The newly created QoS manager first checks, whether it is really talking to console. This is done by sending and expecting a greeting message. Having received anything else, it closes the connection, and exits immediately. The console is informed about duplicated QoS manager invocation and signal-caused termination by sending the appropriate messages. The URL state information is sent in textual form containing the ephemeral port number, the URL and the type of the state change. The console can send QoS requests. These are also in textual form, containing the port number of connection and the actual QoS parameters. The console is also informed when an RSVP event indicates the success or failure of one of its requests.

Communication with the RSVP daemon and broker

The QoS manager communicates with the RSVP Daemon via the RSVP API calls [4]. At each QoS request it probes, whether there is a broker at the remote end. If there is, it creates a new RAPI session, and configures the broker for setting up the other end of RSVP session. The latter is done by sending all the necessary data (such as IP addresses and port numbers, TSpec, etc.) to the proper broker.

5. The QoS broker

The server side requires a small program that interacts in the building of the channel with the requested QoS parameters. This small program, namely the broker, is completely independent from the Web server; nevertheless it acts as a server. The role of the broker is to help „remote controlling" and it does nothing on its own.

The broker should be listening to its „well-known" port until someone tries to connect to this port. Then after an identification it should get all the necessary data (the requested QoS parameters, the server-side port number of the http connection, the client-side IP and port number, etc.) from this „control" connection. With these data the broker is capable of making the first steps in order to build up a channel with the requested QoS (according to the RSVP protocol specification these steps must be made on the sender side [2]). These steps are made with the help of the RSVP daemon. There is at most one „control" connection for each channel with QoS request.

6. Conclusion

This paper describes the proposal for integrating RSVP into the existing WWW architecture, therefore it provides the use of existing Web applications in the Integrated Service Architecture. The proposal is general enough to be used with any Web browsers and Web servers without modifying their code. The implementation of the proposal has been started. At the present the first version of the code is released.

ACKNOWLEDGEMENTS

This work was supported in part by the European Commission under the AC310 ELISA project. The authors are solely responsible for the content of this paper.

REFERENCES

[1] P. White. *RSVP and Integrated Services in the Internet: a Tutorial.* IEEE Communications. Magazine. May 1997.
[2] Braden et al. *Resource Reservation Protocol (RSVP).* IETF RFC 2205. September 1997.

[3] *HTTP 1.1 specification.* Internet RFC 2068.
[4] R. Braden, D. Hoffman. *RSVP RAPI Version 5* draft-ietf-rsvp-rapi-00.ps. June 1997.