Tunnel Agents for Enhanced Internet QoS

Hermann de Meer and Jan-Peter Richter
University of Hamburg
Antonio Puliafito and Orazio Tomarchio
University of Catania

This agent-based approach for improved Quality of Service provisioning follows the open programmable networks paradigm for complementing still-defective Internet reservation schemes. It provides more complete QoS provisioning in a flexible, highly scalable manner. The authors' Java-based agent platform might work especially well in heterogeneous environments, which distributed multimedia systems are most likely to face.

Heterogeneous networking will soon become a reality. Even though new technologies, such as asynchronous transfer mode (ATM), inherently guarantee Quality of Service provisioning, they will continue to use protocols from the TCP/IP world. Consequently, the Internet world needs QoS provisioning.

System developers recently have proposed Internet extensions, such as the resource reservation protocol (RSVP),[1,2] to guarantee real-time services while simultaneously relaxing requirements for reliable communication. Unfortunately, network resources, such as routers, have yet to fully support RSVP reservation, and it is unlikely all future routers will support it. Furthermore, their rather basic functions fail to provide more application-oriented support for QoS provisioning.[2] (For more information, see the "Quality of Service" and "RSVP" sidebars.) A more complete solution would require additional effort.

We propose using agent technologies based on open programmable networks.[3] Though tailored to complement RSVP as a defective Internet protocol-based reservation mechanism, the agent approach also promises to complement incomplete reservation techniques that might arise in the future. Software agents offer an attractive foundation for these networks because they are already arranged to operate in a distributed manner, and they permit flexible communication and cooperation schemes.

Our procedure relies on distributed mechanisms that implement cooperative monitoring techniques to

- monitor system behavior and determine the performance level,
- use system-status information to establish a QoS negotiation phase and distribute application requirements on system resources, and

Quality of Service

The deployment of multimedia services in data telecommunication networks has introduced the concept of Quality of Service to data networking. Transmitting time-related data, such as digital audio and video, demands stringent timeliness in the process of data transmission but usually allows occasional data loss or corruption. Therefore, in a network that supports QoS, a *QoS negotiaton* is part of the connection setup procedure.[1] *QoS parameters*—such as data error rates, packet loss rates, throughput, end-to-end delay and delay variation (delay jitter)—usually express QoS for data-transmission services.

In a *QoS contract* between a service user and a service provider, the service provider commits itself to outperform the agreed-on limit given that the service user imposes a load not exceeding an agreed-on traffic specification.[1] To achieve this, the service provider reserves a sufficient share of its resources for the connection under concern. In addition, it can monitor the user data traffic to comply with the traffic specification, and if the agreed-on QoS cannot be achieved, it indicates this to the user, and a media self adaptation, such as a change in the mode of compression, can be initiated.

Reference

1. J.-P. Richter and H. de Meer, "Towards Formal Semantics of QoS Support," *Proc. 17th IEEE Inforcom '98*, IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 472–479.

RSVP

RSVP is a framework for resource reservation and QoS provisioning mechanisms within the Internet. The framework is an extension of classical IP routers, including RSVP-message processing as well as a more controlled packet-forwarding operation. It is designed to operate inside non-fully RSVP-based networks. If non-RSVP-capable nodes belong to the path of an established session for which the user wants a certain QoS, RSVP tries to reserve adequate resources on the nodes where this is possible and relies on a best-effort strategy in the remaining cases. Of course, in such cases no global guarantees can be given.

The RSVP protocol merely defines the exchange of control messages to build up and maintain a shared knowledge of the reservation state; however, several QoS-supporting *integrated services* are defined within that framework, complementing the protocol with detailed definitions of router behavior. Two such services are controlled-load and guaranteed. The integrated-services working group characterizes the former service as follows:

> Controlled-load service provides the client data flow with a quality of service closely approximating the QoS that same flow would receive from an unloaded network element, but uses capacity (admission) control to assure that this service is received even when the network element is overloaded.[1]

While this form of QoS provision might be sufficient for self-adapting applications, more advanced reservations schemes are necessary if giving QoS guarantees. The latter service is characterized as follows:

> Guaranteed service provides firm (mathematically provable) bounds on end-to-end datagram queueing delays. The service makes it possible to provide a service that guarantees both delay and bandwidth.[2]

Our agent-based approach deploys a guaranteed service within this RSVP framework.

References

1. J. Wroclawski, "Specification Controlled-Load Network Element Service," RFC 2211, Sept. 1997; http://www.isi.edu/rfc-editor/rfc.html.
2. S. Shenker, C. Patridge, and R. Guerin, "Specification of Guaranteed Quality of Service," RFC 2212, Sept. 1997; http://www.isi.edu/rfc-editor/ rfc.html.

- interact with reservation entities within the RSVP routers.

We try to exploit the interaction among properly defined software entities—continuously running inside the system—to implement QoS monitoring, reservation, and adaptation capabilities.[4]

The tunnel problem

Introducing RSVP requires changing the operation of IP routers, while some routers within the Internet will never support RSVP. The RSVP protocol that handles the exchange of control messages simply ignores non-RSVP routers (routers unable to support RSVP). It forwards RSVP messages through a cloud of non-RSVP routers and merely performs reservation outside the cloud. However, because non-RSVP routers can degrade the perceived QoS in an uncontrolled way, there is no end-to-end guarantee. RSVP application on the remaining part of the path is still considered beneficial, because the performance of the non-RSVP routers might sufficiently fulfil the demanded QoS level.

Aside from communicating RSVP control messages through such a cloud of non-RSVP routers, further action is not taken within the RSVP framework to handle such a situation; RSVP tells the application that the QoS values are only approximate.

Java language

Sun devised Java—its object-oriented, portable, interpreted, multithreaded programming language—to develop safe, robust applications in heterogeneous network environments.[1,2]

Adopting a two-stage code-generation process achieves independence from the underlying architecture. A Java compiler compiles the source code, which does not generate a *machine code* directly but a so-called *bytecode*—a high-level architecture-independent code for the hypothetical machine that the Java interpreter and its runtime system implement: Java Virtual Machine. A Java interpreter can then execute the bytecode on any machine.

Sun produced the JVM for a number of platforms, thus ensuring that a Java application can be executed on any platform on the market. Architecture-independence is not only useful for applications used in networks, but also for problems related to distributing software on different platforms: a programmer no longer has to worry about developing various versions for different machines and operating systems.

Modern network-based applications typically need to do several things at the same time. The Java multithreading capability provides the means to build applications with many concurrent threads of activity. Although recent operating systems provide libraries for *multithreading*, their difficulty and dependence on a particular architecture have prevented many programmers from using them. Java, on the other hand, supports multithreading at the language level, providing sophisticated synchronization primitives based on the concepts of *monitor* and *condition variables*.

A basic concept of the Java architecture is that of *class loaders*, which allow the Java runtime system to load classes without any knowledge of the underlying file system. All JVMs include a default class loader that loads the classes of the local file system, but Java lets users define their own class loader, thus overriding the behavior of the default loader. This makes it possible to load classes from various sources—for example, in applets, which load classes from a Web server using the http protocol.

Java 1.1 has introduced an interesting feature called *object serialization*, which lets users take an object and write its state to a byte stream. Deserializing the sequence of previously saved bytes restores the object. By means of this mechanism, users can implement object persistence: an object's lifetime is not determined by whether or not a program is executing, because it can live between different program invocations.

References

1. K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading, Mass., 1996.
2. M.A. Hamilton, "Java and the Shift to Net-Centric Computing," *Computer*, Vol. 29, No. 8, Aug. 1996, pp. 31–39.

Our approach deals with *tunnels* that lead through a so-called cloud of non-RSVP-capable network entities. We define a tunnel as a set of routers constituting the subpath within a cloud of non-RSVP routers; a *tunnel router* refers to a single router within the tunnel. We aim to improve the end-to-end quality by monitoring the tunnel and providing feedback to enhance the reservation scheme.

An agent-based approach

The crucial part of our reference scenario is the tunnel representing an entity with nonpredictable QoS behavior. Adequately monitoring tunnel behavior spreads information throughout the system, carrying out network adaptation to compensate for excessive QoS budget consumption within the tunnel. Agent technology works well because of its high-level flexibility and interaction, as well as its inherent capacity of remote process execution.

We can define an agent as a software module, eventually equipped with AI mechanisms, capable of solving—autonomously or in cooperation with other agents—a certain problem or carrying out a particular task.[5,6] Because an agent should be autonomous, competent, and reliable, it is required to

- operate without direct human action and perform a certain level of control over its own actions,
- properly manage a given situation and improve its behavior by accumulating experience in specific situations, and
- guarantee that its evolution will lead to choices and behavior that will conform to user preferences.

Some researchers—especially in the field of AI—claim that there are other desirable characteristics, such as personality and adaptivity. Mobility, however, is probably the most important.[6,7] Mobile agents—agents capable of migrating from one machine to another within a heterogeneous computer network—easily and effectively help build distributed applications. Some areas widely applying such technology are

- *distributed management*: mobile agents let system designers delegate management functions to remote nodes, reducing the workload on the central station and improving exploitation of available resources;[8]
- *distributed computation*: with the ability to pilot the node on which mobile agents execute, agent technology offers a new paradigm for parallel computation in a distributed network of workstations; and
- collaborative applications: mobile agents support

sharing data and any kind of documents by offering a flexible architecture and letting users share various network resources.

Software-agent functions

We chose an agent-based infrastructure to manage QoS, because it effectively implements adaptation strategies to rearrange available resources and maintain user satisfaction. These strategies exploit agents' cooperation abilities for optimization purposes.[9] Although each agent locally performs reasoning and decision-making processes, agents coordinate themselves by communicating with each other about certain events and internal status information. Consequently, each agent influences every other agent.

To solve the tunnel problem, we need software agents that can

- monitor system status by combining and filtering data to reduce message overhead between managed components and management applications,
- cooperate with other agents by exchanging data and synchronization messages, and
- migrate from one node to another and execute remotely with transparent use of the available hardware and software resources.

At the University of Catania, we have developed a computing platform to help implement these functions, particularly agent retrievability and mobility (available at *http://sun195.iit.unict.it/MAP*).[10] The platform provides a sort of homogeneity among different hardware and software platforms, adopting Java as its underlying framework. Java allows modular system development: users can easily write their own agents from the classes the platform provides.[11] Java guarantees code portability across an ever-increasing number of software and hardware platforms. (See the "Java language" sidebar for more information.)

We developed a specialized version of this platform for QoS management, comprising specific nodes that collect QoS applications.[12,13] We identify such nodes as servers of applications and assume they store QoS applications we can download and execute on demand.

Reference architecture

Figure 1 depicts our reference architecture. A user on the client system interacts with the QoS management component through an appropriate *user agent*. This agent is specific for each service the user requires, and users can specify and negotiate desired QoS parameters.

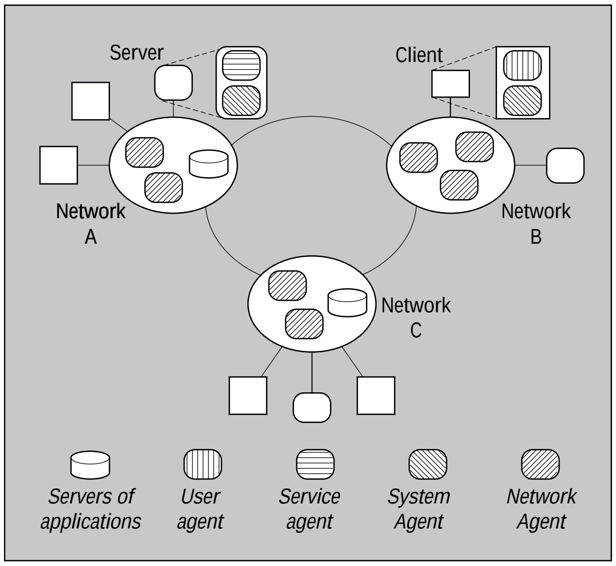

Figure 1. Our reference architecture in which we classify software agents according to their specific functions.

The *system agent* provides information about the client's system state; it relies on the services provided by the host machine's operating system to control the QoS on the client. The *service agents*—present on each server node—monitor whether the QoS parameters (for each service the clients require) are respected. It keeps track of the users logged in and their occupied resources to optimize available resources. This agent also maintains the knowledge of the server's capabilities.

Network agents implement the control functions by

- reserving suitable network resources,
- monitoring system parameters,
- initiating corrective QoS adaptation activities if violations are detected, and
- interacting with the network management system to obtain additional resources and optimize their use.

Tunnel agents

The primary goal of *tunnel agents* is to monitor tunnel properties, interacting with the RSVP routers for assuring the end-to-end QoS to the user.

Tunnel setup

When an application on a receiving node requests a reservation, RSVP carries the request through the network, visiting each node belonging to the routing path between the receiving machine and sender. On each intermediate node, RSVP tries to reserve adequate resources to guarantee the desired QoS. If there are enough resources on each router, and if the admission test succeeds, the data stream starts flowing on the preestablished path with the negotiated QoS. However, some non-RSVP nodes, referred to as a tunnel, might be located along the route (see Figure 2).

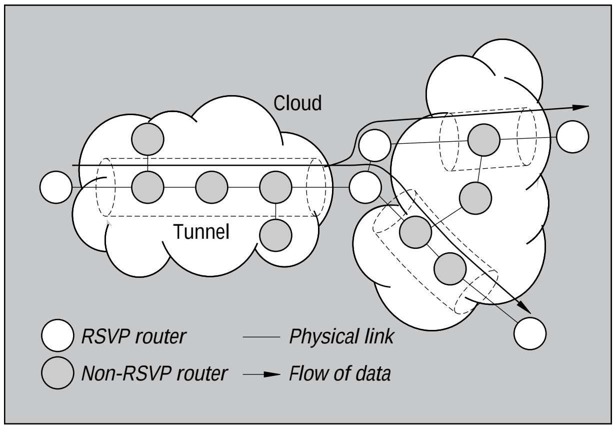

Figure 2. An example of non-RSVP routers constituting tunnels.

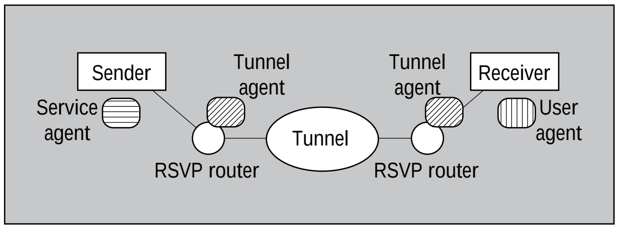

Figure 3. Tunnel agents located on the RSVP nodes at each tunnel's edge.

Our agent approach monitors the tunnel behavior and obtains performance indices to characterize the delay in tunnel crossing. Two particular network agents—*tunnel agents*—are associated with each tunnel. Figure 3 shows tunnel agents located on the RSVP nodes at each tunnel's edge—we refer to these nodes as *edge routers*. An RSVP router can be an edge router for two adjacent tunnels, if surrounded by non-RSVP routers. In this case, two instances of a tunnel agent are located on the edge router. We assume that each RSVP node is able to execute our agents.

Furthermore, we assume all nodes, including non-RSVP nodes, support standard network management protocols, such as SNMP, to provide some form of monitoring.[14] In an extreme scenario, none of the routers support RSVP and the tunnel spans over the entire path. In this case, the tunnel agents are located on the application hosts and provide the application with useful performance-monitoring data to adopt traffic volume and encoding. Tunnel agents running on edge routers are not resource-consuming, although the edge routers are already performing RSVP functions. The tunnel agents simply manage infrequent events, while a router continuously performs packet forwarding.

A generic node, playing the role of a QoS-application server, maintains the agent code. Once a tunnel is identified, the user agent located on the receiving application node sends a request to the nearest QoS-application server to upload the tunnel-agent code on the edge router, where it then dynamically executes.[10]

This approach avoids code installation on each node: a tunnel's location depends on dynamic factors, such as the existence and location of the session path, as well as more static properties, such as the nodes' RSVP capability. A node might never be at the edge of a tunnel. A tunnel agent installed on such a node would not be useful and would only consume node resources. The existence of simple mechanisms for code retrieval and downloading is thus a useful feature for our specific problem.

Our approach also exploits agent migration when the Internet's dynamic routing algorithms relocate the session path. When an already instantiated tunnel agent migrates, it carries some knowledge of flow requirements to the new node.

SETUP PROCEDURES

RSVP packets are sent as raw IP datagrams with the path end-point destination IP address. Each RSVP Path message and Resv message also carries the last RSVP router's IP sender address. RSVP packets are slightly modified each time they cross an RSVP router; the TTL field of the IP header (IP-TTL) is copied into the RSVP header's `Send_TTL` field. A non-RSVP router does not modify an RSVP message but forwards it to the following router along the path with a decremented IP-TTL. If a receiving RSVP router discovers a mismatch between IP-TTL and `Send_TTL`, it assumes there is at least one non-RSVP router between the last RSVP router and itself. It then is aware of a tunnel and of the edge router's IP address at the other side of that tunnel.[15] When the receiving application becomes aware of non-RSVP routers along the path, its user agent initiates the tunnel setup procedure.

First, the agent must determine the exact tunnel location, using one of the following methods.

- Query the RSVP routers, using SNMP, for their respective RSVP neighbors and direct IP connectivity. If an RSVP router has no direct IP connectivity to one of its neighbors, it is an edge router. The user agent then follows the chain of RSVP neighbors to learn about the RSVP and edge routers along the path. Because an SNMP request or reply involves only two UDP (user datagram protocol) packets, it is the best solution for small networks. However, because this method scales poorly, it is more appropriate for short end-to-end paths.
- Use a *discovery agent* sent by the receiving-application host. This agent migrates among RSVP routers and issues local SNMP requests, learning about the RSVP-neighborhood relation and the directly connected routers. Finally, the agent returns the collected information to the application host. This solution needs only linear time and is therefore more suitable for longer end-to-end paths.

Network management

Network management can be defined as a collection of additional functionalities of a data telecommunication network that enables the management of that network.[1] A *network management protocol*, such as Internet's *simple network management protocol* (SNMP), exchanges the necessary control information to manage network-operation aspects—for example, failures, performance, configuration, security, and accounting. Management actions are carried out by control communication between a remote *network management station* (control console) and several *network management agents* that have direct access to the relevant software and hardware components of the network.

Unlike the mobile agents, network management agents are hardware-dependent, colocated with the components they make accessible, and thus immobile. To logically organize the manageable information in a network management system, the managed objects are organized in a conceptual database called *management information base* that is subdivided into individual MIBs for individual software and hardware components. With the abstraction of a conceptual database, network management actions can be interpreted as query and update operations in a distributed database system.

References

1. W. Stallings, "SNMP, SNMPv2, and CMIP," *The Practical Guide to Network-Management Standards*, Addison-Wesley, Reading, Mass., 1993.

- Extend the RSVP code installed on the RSVP routers. Because each RSVP router knows whether or not it's an edge router, each could send a message to the application host. However, this requires extending the functionality provided by the RSVP module on each router, which can be difficult.

Selecting either the first or second tunnel-discovery method can be done adaptively, based on the number of RSVP hops.

After determining the tunnel location, a tunnel-agent instance is uploaded to each edge router from a server of applications. Each tunnel agent located on an upstream edge router starts a trace-route procedure to learn about the non-RSVP routers constituting its tunnel. This information is communicated to the corresponding downstream tunnel (a multicast tunnel can have more than one downstream edge). All tunnel agents monitor the tunnel behavior and eventually interact with the RSVP protocol.

TUNNEL AGENTS INTERACTING WITH TUNNELS

Tunnel agents must make decisions on how to achieve network reconfiguration. These decisions are based on information concerning the tunnel structure and state. The tunnel agents determine the topological structure of the tunnel—the set and connectivity of non-RSVP routers—using a trace-route procedure and SNMP, as described above. They also use SNMP to monitor basic performance measures of the tunnel routers, such as interface utilization, discard rates, or queue lengths, by polling the respective MIB (management information base) entries.

Additionally, the set of agents at the tunnel's edge use cooperative monitoring techniques to monitor traffic going through the tunnel and its received QoS (see the "Network management" sidebar). For example, a pair of tunnel agents can measure tunnel-end-to-tunnel-end delay by sending time stamped probe messages. Messages, which are time stamped at the sender side of the entire path, cannot measure tunnel-end-to-tunnel-end delay without additional message exchange between the tunnel agents.

Inter-agent communication is not as sensitive to QoS degradations as the realtime user data. Because control traffic is mostly sensitive to loss, well-established methods of timeout and retransmission in the Internet can protect it. A slower transmission of a control packet or the additional delay caused by its retransmission in case of a loss will cause the adaptation mechanism to work more slowly. However, after a short time the adaptation will be complete, limiting the QoS degradation to a *transient* one. Using forward-error correction techniques also enhances control-traffic transmission. By using a redundant encoding scheme, we can tolerate a limited number of losses. Because the bandwidth demand of the control traffic is much lower than that of user data, this additional traffic is not likely to further enhance network congestion.

Much of the control traffic will be exchanged between the (last) downstream tunnel agent and the user agent on the receiving node, as well as between tunnel agents of neighboring tunnels. Because this traffic runs entirely over RSVP-capable nodes, it can be highly prioritized and thus protected from QoS degradations.

TUNNEL AGENTS INTERACTING WITH RSVP

The total end-to-end delay is the sum of all delays encountered along the transmission path. Therefore, a bound on end-to-end delays required by an application is considered as a QoS budget that the involved intermediate node can arbitrarily spend.

Because the RSVP-guaranteed service relies on a fluid-flow and leaky-bucket approach, a reservation of a minimum service rate R is used to distribute the delay budget. Each RSVP router reserves resources sufficient to support rate R. This sets a bound on each local queuing delay, which bounds the sum of all local queuing

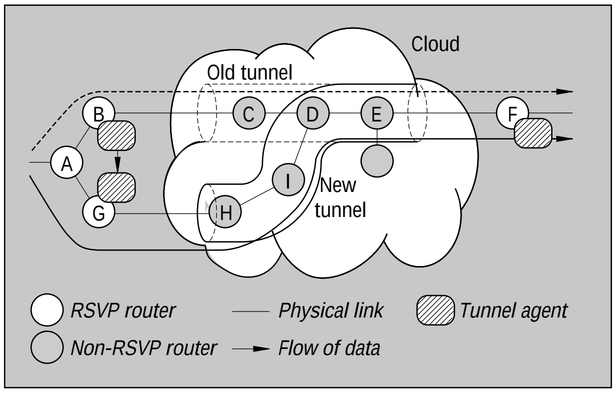

Figure 4. Agent migration after route change.

delays to a given value.[16] The receiver calculates R so that the resulting end-to-end delay meets the QoS demands. However, without agent support this calculation would turn on the assumption that all routers along the path were RSVP-capable—even if it detects non-RSVP routers.

Our agent-based approach uses a higher rate, $R' = R + d$, to set aside some additional delay budget not distributed among the RSVP routers. The user agent calculates R' in cooperation with the application and the state information provided by the tunnel agents. The additional delay budget, characterized by the difference between the original rate R and the announced rate R', is subsequently also communicated to the tunnel agents. If more than one tunnel is present along the path between a sender and a receiver, all tunnel agents agree on how to distribute the additional delay budget, using a distributed inter-agent negotiation procedure.

By monitoring the tunnel behavior, the agents verify if the QoS budgets currently consumed by the tunnel matches the value assigned to that tunnel. In case of a mismatch, the tunnel agent initiates horizontal renegotiation. First, it renegotiates the partitioning of the set-aside delay budget by reiteration of the interagent negotiation procedure. If there is only one tunnel or if the inter-agent renegotiation fails, it renegotiates the QoS parameters in the RSVP routers along the path. It also communicates the additional need for a delay budget to the user agent, which recalculates R' and informs the application to change its RSVP reservation messages accordingly.

Reassigning budgets attempts to compensate for the adverse effect of the tunnel routers. Of course, the opposite is also true: if traffic fluctuation enhances the tunnel, it initiates the horizontal renegotiation to relax unnecessarily stringent requirements on the RSVP routers.

The renegotiation phase can fail due to a lack of resources. In this case, the standard RSVP protocol mechanism sends an indication to the application; in turn, the application can adapt to the lower-level QoS or simply terminate.

TUNNEL-AGENT MIGRATION

Routing in the Internet is a dynamic process, and a route can change during connection time, which can exploit an agent's ability to migrate between routers. For example, in Figure 4 a stream of data flows from routers A through F. When the connection between C and D breaks, the usual routing algorithms of IP-based networks find an alternative route—via G, H, and I, for example. The RSVP-protocol mechanisms relying on IP routing also follow this alternative route and establish a reservation state along the new path.

The tunnel agent located on B will notice that it is no longer at the edge of an active tunnel and will initiate a relocation procedure. This procedure operates in cooperation with the agent on the other end of the tunnel or with the user agent on the application host—depending on whether the upstream or downstream end of a tunnel has moved. Applying algorithms similar to the tunnel-detection procedure discovers the new tunnel topology. If only one edge of the tunnel has moved, the tunnel agent initiates its own migration to the new tunnel-edge location. If RSVP routers are located along the new path between G and F, the new tunnel actually consists of a sequence of tunnels. Then new tunnel agents must be started in addition to the migration of one tunnel agent.

In Figure 4, tunnel routers D and E both belong to the old route as well as the new route and therefore constitute part of the new tunnel. The tunnel agent migrates from routers B to G to save information about the old tunnel, and it reuses that information when managing the new tunnel. It can keep topological information and, to a certain extent, load-information on the unmodified parts.

Of course, a route change can also cause a tunnel to completely disappear. If the tunnel agents detect they are now connected by a continuous chain of RSVP routers, they inform the other tunnel agents along the path—if there are any—as well as the user agents on the application hosts, and terminate.

Implementation issues

We use Java as the programming language for our agents because of its modularity and portability features, ability to dynamically load classes from different sources, and object-serialization capabilities.[17] (See the "Java language" sidebar.) We make extensive use of these basic mechanisms, because they support the procedures for tunnel identification and tunnel-agent migration inside the network.

Agent mobility

Mobility allows tunnel and discovery agents to move from one router to another to achieve their goals. One feature lets an agent move with its state, resuming execution from the point where it was interrupted before the migration.

An agent in execution consists of the code (*program state*), variables content (*data state*), and stack (*execution state*). In general, a complete migration requires transfering all three components. Yet, even if program and data state can be managed with a limited effort, the capture, transfer, and restoration of the execution state can cause considerable problems. In particular, in the case of an interpreted language such as Java, the interpreter state includes a portion of the execution state, and capturing this state without modifying the interpreter is practically impossible. The specific application in question gives the agents a high level of competence, so migration never initiates from outside but is started by the agent itself when specific conditions occur.

For example, consider the mechanisms that identify tunnels through discovery agents, and tunnel-agent migration already instantiated due to relocating the session path. Both cases require agent-migration capabilities and status information. More specifically, a discovery agent has a list of routers already visited and indicates with a flag whether or not the router supports RSVP capabilities; after a route reconfiguration, a tunnel agent needs information characterizing routers within the unchanged part of the original tunnel. Because the agent is responsible for the migration, it can save information required for correct execution resumption within some of the object variables. Because these variables belong to the data state, they can be transferred and later restored when the agent execution is resumed in the site of destination.

The mobility of tunnel and discovery agents is based on the *object serialization* facility present in Java (version 1.1). Object serialization can represent an object as a stream of bytes. It saves all the object variables within the serialized object representation, together with the references to objects contained within it.

When an object is serialized, Java does not store the bytecode of the class to which the object belongs in the byte stream. Thus, because in our system the classes needed are not always in the destination nodes, we had to exploit another Java characteristic: the ability to load some classes dynamically at runtime from different sources.

We used the NetworkClassLoader to load the classes necessary for agent execution. If the class needed is not present locally, the NetworkClassLoader searches for it within the nodes contained in a list specified during the configuration and the startup of routers, but that can be updated at runtime, thanks to some information brought by the agents. Such a mechanism effectively exploits the network. Transferring the class bytecode occurs only when required (when several instances of the same agent migrate, we do not need to transfer the same class); in any case, we can also perform the transfer from the closest node—not necessarily from the original agent node.

This transfer feature helps once our system reaches a steady-state condition, because the majority of the classes are already present on the routers and downloading actions do not occur very often. Furthermore, this mechanism avoids downloading onto routers huge amounts of software that might not be used. It is also easy to support extensions for new types of agents: we can automatically download classes that new agents might require, thus guaranteeing future system extension.

Communication mechanisms

Our system also supports an agent's ability to communicate with other agents. The communication among the agents takes place through the exchange of synchronous or asynchronous messages. This solution (unlike some mechanisms based on RPCs) is flexible and permits implementation of several schemes of communication and synchronization among agents.

Encoding and transmitting messages takes place in the same way as agent migration. In fact, the mechanism selected for transferring messages is object serialization, which allows complex objects to be sent among different agents.

With a synchronous message, we obtain a behavior similar to an RPC. The sending agent calls a communication primitive, passing the message and destination-agent identifier as parameters, and then it waits for a reply message. Conversely, in the case of an asynchronous message, the sending agent invokes another primitive and, after sending the message, continues with its execution.

Asynchronous communication is used for interaction among tunnel agents. Once a tunnel has been identified, the tunnel agents on the edges communicate with each other, sharing information concerning the tunnel behavior. To retrieve information from the routers within the tunnel, a tunnel agent uses synchronous communication. Sending a query to the local SNMP agent implements this retrieval.

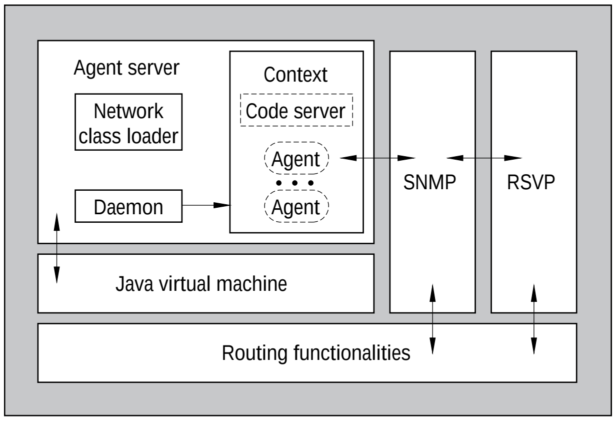

Figure 5. Software modules required on a router.

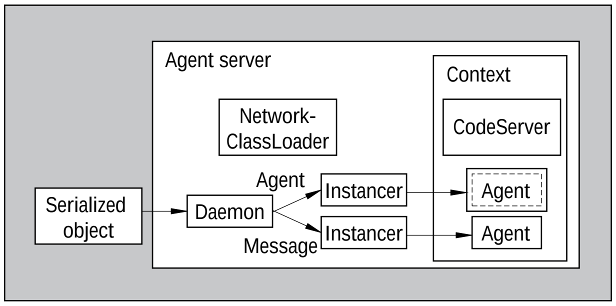

Figure 6. An agent server.

Agent-execution environment

We now review the software environment that must be present on a router to allow agent execution. Figure 5 shows the logical organization of a router's main functional entities. With routing functionalities, we indicate the basic mechanisms for routing packets, which depend on the router itself. These mechanisms can be implemented directly in hardware or emulated through adequate software. In the current implementation state, a software approach executes routing functionalities; we use simple PCs configured as routers, running Unix FreeBSD-2.2.5 (available at *http://www.freebsd.org*).

We require an *SNMP agent* to monitor and retrieve information about the status of the router and the different connections going through it. The *RSVP module* lets the router eventually use this protocol to communicate with other routers for resource-reservation purposes. It currently uses RSVP version 1, as implemented by the ISI rsvpd Release 4.2a1 (available at *http://www.isi.edu/rsvp/*), and the packet scheduler mechanism, provided by ALTQ version 0.4.1 (available at *http://www.csl.sony.co.jp/person/kjc/software.html*).

An agent server is also instantiated on a router, which represents the agents' execution environment. Because these agents are written in Java, a Java Virtual Machine is active on the routers using JDK 1.1 (Java Development Kit) for FreeBSD (available at *http://www.freebsd.org/java/*). The reference structure depicted in Figure 6 helps analyze the agent-server components.

The *daemon* constantly listens to an input port, waiting for tunnel or discovery agents to migrate from other nodes or receiving messages to be delivered to local agents. Both messages and agents travel in a serialized form. Each time a stream arrives from the network, the daemon creates a specific entity called *Instancer*, which instances the serialized object, whether it is an agent or a message. In both cases, it passes the object, once instanced, to the *Context*, which makes it run (if it is an agent) or sends it to the receiver agent (if it is a message).

The Context object knows all the agents currently instantiated on the node and provides all the functionalities needed for their management. It owns specific methods for creating, executing, suspending, deactivating, reactivating, and killing an agent. Each serialized object arriving from the network always passes to the Context. If this object is an agent, the Context initializes it, providing a reference to the Context itself: it then starts its execution and updates the list of the agents running locally. If the object is a message, the Context will deliver it to the destination agent, after verifying its ability to receive the message.

The NetworkClassLoader lets tunnel agents execute on an agent server even if their class is not present. When a class referenced by the agent is not locally present, the NetworkClassLoader downloads it from one of the agent servers indicated in a list that the Context manages. The class is then stored in the global cache of the Context, and it is then accessible from the other agents currently executing on the server.

The Context also contains an object named *code server*, dynamically created each time a NetworkClassLoader requests a class. Its goal is to make the requested classes accessible if locally present. The Code Server contains a table listing the classes available on the node.

Implementing QoS measurements

For the tunnel agents to carry out the QoS measurements, the transport protocol should include mechanisms supporting this task. Therefore, we designed and implemented a simple, yet flexible transport protocol for realtime user data. The Protocol Data Units (PDUs) of the *Flexible Continuous Media Transfer Protocol* can be configured to contain timestamps or redundant data, such as sequence numbers or checksums, to easily monitor delays, loss, and error rates. Because we can also use these protocol mechanisms for the usual protocol functionalities, we are investigating an integrated concept focused on continuous media transmission. The design of the protocol, however, is beyond the scope of this article.

Our approach, based on intelligent software agents, complements still-defective reservation schemes, such as RSVP in the Internet. We can thus achieve more complete QoS provisioning in a flexible and highly scalable way. Future telecommunication networks will be increasingly open, so we will be able to deliberately add services such as QoS-management functionalities. A Java-based agent platform seems particularly promising in a heterogeneous environment, which distributed multimedia systems will probably face. We are still investigating our system's implementation efficiency. In particular, we are trying to determine agent-control strategies to avoid oscillation effects and provide limits for timely agent response.

REFERENCES

1. L. Zhang et al., "RSVP: A New Resource Reservation Protocol," *IEEE Network*, Vol. 7, No. 5, Sept. 1993, p. 8.
2. R. Braden et al., "Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification," RFC 2205, Sept. 1997; http://www.isi.edu/rfc-editor/rfc.html.
3. A. Lazar, "Programming Telecommunication Networks," *Proc. 5th Int'l Workshop on Quality of Service*, Chapman & Hall, New York, 1997, pp. 3–22.
4. H. Hafid and G.V. Bochmann, "Quality of Service Adaptation in Distributed Multimedia Applications," to be published in *ACM Multimedia Systems J.*, Vol. 6, 1998.
5. D.M. Chess, C. Herrison, and A. Kershenbaum, "Mobile Agents: Are They a Good Idea," *Mobile Object Systems: Towards the Programmable Internet, Lecture Notes in Computer Science*, No. 1222, Springer-Verlag, Berlin, Apr. 1997, pp. 25–48.
6. M.R. Genesereth and S.P. Ketchpel, "Software Agents," *Comm. ACM*, Vol. 37, No. 7, July 1994, pp. 48–53.
7. O. Etzioni and D.S. Weld, "Intelligent Agents on the Internet: Fact, Fiction, and Forecast," *IEEE Expert*, Vol. 10, No. 4, Aug. 1995, pp. 44–49.
8. Y. Goldszmidt et al., "Decentralizing Control and Intelligence in Network Management," *Proc. Fourth Int'l Symp. Integrated Network Management*, 1995.
9. S. Fischer and H. de Meer, "Decision Support in Cooperative QoS Management," *Proc. Sixth IEEE/IFIP Int'l Workshop Quality of Service*, IEEE CS Press, 1998, pp. 247–255.
10. A. Puliafito, O. Tomarchio, and L. Vita, "A Java-Based Distributed Network Management Architecture," *Third Int'l Conf. Computer Science and Informatics*, Paul Wang, 1997.
11. K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading, Mass., 1996.
12. A. Puliafito, O. Tomarchio, and H. de Meer, "An Agent-Based Framework for QoS Management," *First World Congress on System Simulation WCSS'97: Fourth ANMT on QoS*, Soc. for Computer Simulation, Singapore, 1997, pp. 392–396.
13. H. de Meer, A. Puliafito, and O. Tomarchio, "Management of QoS with Software Agents," *Cybernetics and Systems*, Vol. 29, No. 5, 1998, pp. 499–523.
14. W. Stallings, "SNMP, SNMPv2, and CMIP," *The Practical Guide to Network-Management Standards*, Addison-Wesley, 1993.
15. F. Baker, J. Krawczyk, and A. Sastry, *RSVP Management Information Base*, RFC 2206, Sept. 1997; http://www.isi.edu/rfc-editor/rfc.html.
16. S. Shenker, C. Patridge, and R. Guerin, "Specification of Guaranteed Quality of Service," RFC 2212, Sept. 1997; http://www.isi.edu/rfc-editor/rfc.html.
17. M.A. Hamilton, "Java and the Shift to Net-Centric Computing," *Computer*, Vol. 29, No. 8, Aug. 1996, pp. 31–39.

Hermann de Meer is an assistant professor at the University of Hamburg and a visiting professor at Columbia University. His research interests include distributed systems, multimedia, QoS, distributed AI, and selected areas of cognitive science. He received both his MS and PhD in computer science from the University of Erlangen-Nürnberg. Contact him at the Dept. of Computer Science, Univ. of Hamburg, Vogt-Kölln-Str. 30, 22527, Hamburg, Germany; demeer@informatik.uni-hamburg.de.

Antonio Puliafito is an associate professor of computer engineering at the Institute of Computer Science and Telecommunications, University of Catania. His research interests include performance and reliability modeling of parallel and distributed systems, networking, and multimedia. He received his EE from the University of Catania and his PhD in computer engineering from the University of Palermo. He is coauthor (with R. Sahner and K.S. Trivedi) of *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package* (Kluwer). Contact him at the Inst. of Computer Science and Telecommunication, Univ. of Catania, Viale A. Doria 6, 95025 Catania, Italy; ap@iit.unict.it.

Orazio Tomarchio is a PhD candidate in computer engineering at the University of Palermo. His research interests include network computing, mobile agents, distributed object-oriented programming, and network management. He received his MS in computer engineering from the University of Catania. Contact him at the Inst. of Computer Science and Telecommunication, Univ. of Catania, Viale A. Doria 6, 95025 Catania, Italy; tomarchio@iit.unict.it.

Jan-Peter Richter is a research associate and PhD candidate in the computer science department at the University of Hamburg. He has been working in the areas of distributed operating systems and data networking, focusing on problems of QoS support and measurement. He received his MS in computer science from the University of Erlangen/Nürnberg. Contact him at the Dept. of Computer Science, Univ. of Hamburg, Vogt-Kölln-Str. 30, 22527, Hamburg, Germany; richter@informatik.uni-hamburg.de.