5

A Multi-Agent Architecture for Cooperative Quality of Service Management

Abdelhakim Hafid[1] *and Stefan Fischer*[2]

[1]*Computer Research Institute of Montreal*
Telecommunications and Distributed Systems Division
1801, McGill College avenue, #800
Montreal, Canada, H3A 2N4
[2]*Université de Montréal, Dept. d'IRO,*
C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada

Abstract

The QoS management approaches developed so far are usually not suitable any more for applications which involve too large a number of communicating entities; for instance, negotiation of QoS parameters between the sender and every single receiver becomes impossible (in case of thousands of receivers). To solve this problem, we developed a QoS management approach, we called Cooperative QoS management, which allows a decentralized cooperative management of QoS; it does not limit the number of users of the application. In this paper we present a multi-agent architecture that implements the cooperative QoS management approach. Agents are installed on system components, such as routers and end-systems; when a user asks for a service with specific QoS requirements a kind of cooperation is initiated between agents to ("best") serve the user.

Keywords

QoS, QoS management, agent, multimedia

1 INTRODUCTION

Much work on Quality of Service (QoS) has been done in the context of high-speed networks in order to provide for some guarantee of quality for the provided communication service. More recently, QoS has been considered in a more global context, including also the end systems, such as the user's workstations and database servers. Various global QoS architectures have been developed (for a recent overview see [1]), which include also functions for performance monitoring, resource allocation and QoS management. For instance, in previous work [6], we have developed a framework for QoS management of distributed multimedia applications which stresses two points: (a) the user should define (through a suitable user interface for

Management of Multimedia Networks and Services R. Boutaba & A. Hafid (Eds.)

QoS negotiation) the criteria which are used by the system to select the "best" system configuration for the application at hand, and (b) the selection of an appropriate system configuration is the first step of the QoS management process, followed by resource reservation and commitment, which is performed during the initialization of the multimedia application and each time a QoS renegotiation is required. Renegotiation may be initiated by the user if his/her preferences change, or by the application when some system component does not satisfy the initially agreed QoS characteristics. We showed the feasibility of this approach by implementing it in a prototype system for a remote news-on-demand service [5]. The negotiation process involves three parties: (1) the database server, which contains the meta-information of the documents including all existing variants, (2) the network and (3) the user workstation, which knows the user's preferences and may also impose certain QoS restrictions.

In multimedia applications including multicasting to many users, such as teleconferencing or educational applications, this global QoS management approach which involves a few system components, e.g. as for remote database access of single users, is not workable any more, because the number of users involved is too large (e.g. thousands) for a global management approach. For instance, negotiation of QoS parameters between the sender and every single receiver becomes impossible, since (1) the system would quickly become overloaded and (2) it would have to take into account (and possibly provide) many different qualities requested by users. Instead, a more decentralized approach seems suitable, where QoS management functions such as QoS negotiation, adaptation or renegotiation are distributed over the network. We developed such an approach called *Cooperative QoS Management* [3] (*CQoSM*), where so-called *agents* are installed on the routers and end systems participating in an application. These agents cooperate with each other in order to provide the QoS levels requested by the application. All important QoS functions such a negotiation, adaptation and re-negotiation are supported. As a special feature, the agent communication allows for user cooperation in quality selection: If users cooperate and decide to request a service in the same quality, less resources have to be reserved, which in turn leads to lower communication costs and higher resource availability for other applications. A detailed description of this approach may be found in [3].

In this paper, we present a multi-agent architecture for CQoSM which will later serve as a framework for implementations. In Section 2, we first describe in detail the multi-agent architecture in terms of media source, media sink and router agents. Section 3 further details on one of these agents, namely the media sink agent. In Section 4, we present the protocols which are executed between the agents in order to provide a cooperative solution for all arising QoS problems. Finally, Sections 5 concludes the paper and gives an outlook on future work such as a prototype implementation.

2 A MULTI-AGENT ARCHITECTURE FOR COOPERATIVE QOS MANAGEMENT

Figure 1 presents an architecture of the cooperative QoS management (CQoSM) approach based on the concept of agents. Each component of the system in question is extended with an agent; examples of these components are routers, host machines, and servers. The agents implement the protocols provided by the CQoSM approach for a given application. The architecture shown in Figure 1 is essentially independent from the type of applications and the technologies and software in use; it is applicable for any multimedia system that requires QoS management, such as QoS negotiation and adaptation. This does not mean that the agents have the same implementation code. Rather, an

agent offers an interface which provides a certain number of standard operations, but the implementation of these operations depends on the component, e.g. its technology and the software it supports.

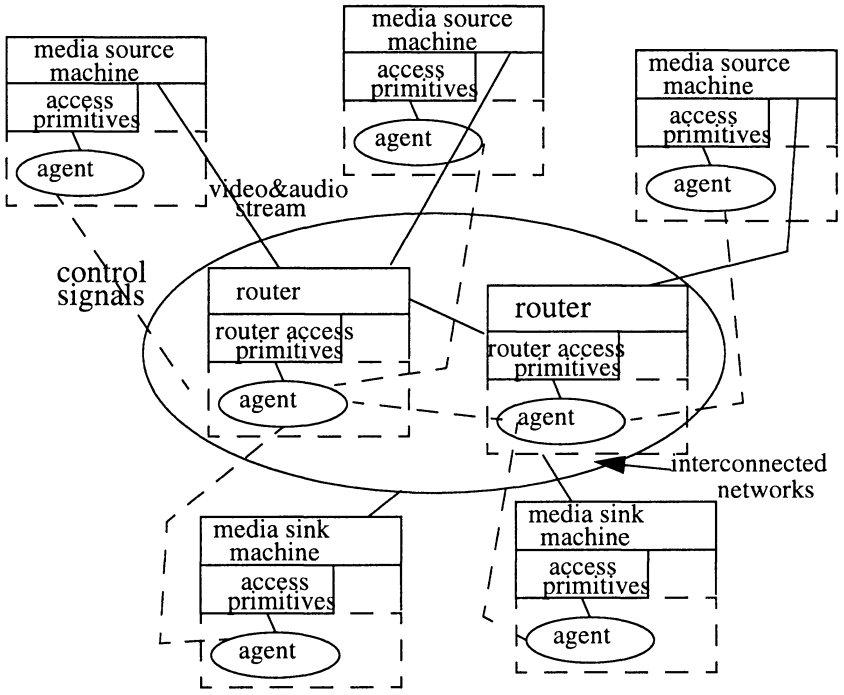

Figure 1 .A multi-agent architecture for CQoSM

The system that supports QoS management for multimedia (MM) applications can be easily extended with new components without code modification of the existing agents; one has only to implement the agents to be installed in the new components. It is obviously imperative that an agent communicates with the component where it is hosted; access primitives allow agents to use abstraction as long as the components agree on the basic language of access. However, a component is free to implement an access primitive in whatever way it sees fit.

We identified three types of agents: media sink agents, router agents, and media source agents. A user participating in a multimedia session, his/her machine can play the role of a media source and/or media sink. Each agent plays a specific role supporting CQoSM.

2.1 Router agent

A router agent is located in any router of the system; to support CQoSM, it maintains a state variable which we call R_Tree_List (Figure 2). Each time a multicast tree is built (for a given application), the router agent located in any router of the tree, creates a new entry in Tree_List; this entry contains (1) the identifier of the tree, Tree_Id; (2) the identifier of the upstream router agent, U_Agent_Id; (3) the identifier of any downstream router agent, D_Agent_Id, with the QoS, Q, the agent does provide; (4) the list of qualities, A_List_QoS, available from the source of

data (the root of the tree in question); and (5) the list of qualities, C_List_QoS, currently available from the agent. A router agent obtains the information about the identifiers of downstream and upstream agents by communicating with the routing protocol in use; this allows a high portability of CQoSM. More specifically, the router agent asks for routing entries from the routing protocols; a routing entry consists of the identifier of one incoming router and a set of the identifiers of outgoing routers.

R_Tree_List

Tree_Ident	Downstrean_agents [(D_Agent_Id,Q), ..]	Upstream_agent (U_Agent_Id)	A_List_QoS [Q1, Q2, ..]	C_List_QoS [Q1, Q2, ...]

Figure 2. State variable of a router agent

The identifier of a router agent corresponds to the identifier (address) of the router; thus, router agents are uniquely identified. We assume that the routing protocol notifies router agents when a multicast tree changes, e.g. a user who leaves or joins the session; this information is necessary for router agents to initiate appropriate negotiation. To make CQoSM work with different routing protocols, a standard interface should be defined; this interface should allow any router agent to get the necessary information to execute appropriate actions.

2.2 Media sink agent

A media sink agent is located in any host machine that implements CQoSM; it maintains a state variable which we call Si_Tree_List; it consists of a list of tuples (Tree_Ident, Upstream_agent (U_Agent_Id)). A media sink agent provides means to the user, via a user interface, to specify (1) the desired QoS he/she prefers to receive from a given sender; that is, the user may select different qualities from different senders for the same session; and (2) the maximum cost he/she willing to pay to be a participant in a session.

The user agent provides also via the user interface a means to start and stop the available MM services; these services are displayed in a graphical window for the user. Each time, the user wants to start a service, the agent invokes a primitive which is provided by a predefined interface to start the service. However, before starting the service the user should specify his/her QoS/cost requirements for each stream he/she receives from the participants in the session. A more detailed description of the media sink agent is presented in Section 4.

2.3 Media source agent

A media source agent is located in any host machine that implements CQoSM; it maintains a state variable which we call So_Tree_List (Figure 3).

The main role of a media source agent is to ask (when appropriate) the source to transmit information with a certain quality. Initially, a media source transmits data with all available qualities, including the best quality. Each time a participant joins or leaves the session, router agents execute the protocols that implement CQoSM; in case all user QoS requirements are less important than the available qualities, the media source agent might ask the media source to deliver only the requested qualities. This can be beneficial

in the case of an application where participants do not join or leave the session frequently; otherwise, the agent operation is not necessary, rather it may introduce some undesirable oscillations.

So_Tree_List

Tree_Ident	Downstrean_agents [(D_Agent_Id,Q), ...]	A_List_QoS [Q1, Q2, ...]	C_List_QoS [Q1, Q2, ...]

Figure 3. State variable of a media source agent

3 MEDIA SINK AGENT

The media sink agent is one the main component of the architecture of CQOSM. Figure 4 shows the main component of the agent. Each client machine of a system supporting CQoSM should contain a media sink agent (MSA). The MSA allows the user to specify his/her requirements in terms of QoS; more generally it allows local QoS management. It also controls the available services, and supports functions that manage application (conference) sessions, such as floor control and membership control.

Let us focus on the functional behavior of an MSA related to QoS management and service control. The description of the session control functions is out of scope of the paper and can be found elsewhere [2].

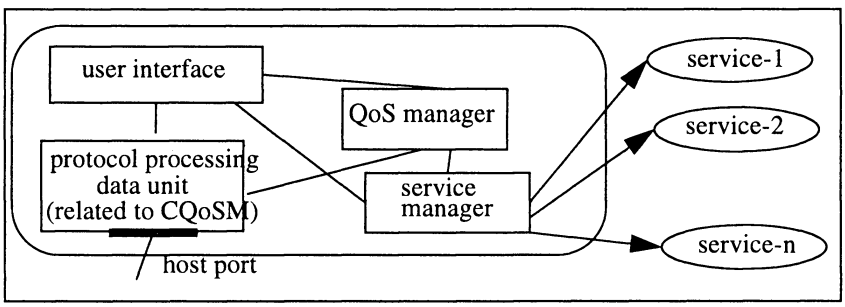

Figure 4. Media sink agent architecture

3.1 User interface

The user interface consists mainly of two parts: QoS interface and service control interface.

The QoS interface allows the user to negotiate and renegotiate his/her requirements in terms of QoS/cost; the user specifies his/her requirements via a graphical user interface [5]. This activity is performed for each service (e.g. video QoS requirements are specified when *vic* is in use, while audio QoS requirements are specified when *vat* is in use). A detailed description of QoS interface in the context of remote access to MM database can be found in [5]; the main parts of this inter-

face can be reused to implement the QoS interface of an MSA.

The service control interface allows users to start or stop a service which is available on their host machines. Examples of these services are vic, vat, sd, and ivs [8]. The interface provides means to control available services; besides start and stop operations, the interface also provides invite, quick, leave, and join operations; other operations, such as floor control, can be also provided.

3.2 QoS manager

The QoS manager provides mainly three functions: local QoS negotiation, QoS mapping, and QoS monitoring.

Local QoS negotiation: The QoS manager checks whether the client machine characteristics, such as the screen size and the screen color, support the user QoS requirements. If the client machine does not support the QoS requested by the user, a rejection (likely with an offer) is sent to the user via the user interface. Then, the user has the choice to abandon the session, accept the offer, or initiate a renegotiation.

QoS mapping: The QoS manager maps the user QoS requirements into relevant QoS parameters for the requested service provider. For example, the network provider does not know how to handle or manage the frame rate and the video resolution parameter; rather, it knows how to handle and manage the throughput parameter (packets/s). Thus, the mapping of frame rate and the video resolution into throughput is necessary to allow the network provider to support the services requested by the user. The mapping functions depend on the service in question; for each available service mapping functions should be provided. These functions can be implemented as (1) analytical functions which is difficult to realize (an example of AAL-ATM mapping QoS parameters can be found in [11], and examples for user-transport mapping are given in [4]); or (2) mapping tables which can be built by processing statistical information gathered during service experimentations.

QoS monitoring: The QoS manager provides means to perform continuous measurement of the QoS which is actually provided, for each running service (e.g, for *vic* and *vat*). This allows to detect and notify any QoS violation: When the measured value of a QoS parameter does not meet the agreed one, a notification is issued, indicating the violation, and preferably the cause. An implementation of the monitoring function in the context of remote access to MM databases can be found in [12].

3.3 Service manager

The service manager controls the available services; primarily, it allows to start and stop the services. More generally, the service manager communicates with the services to perform control functions and to get the information necessary for mapping and monitoring purposes; for example, the QoS manager may collect feedback reports of RTCP [9] (in the case of vic and vat services) to react to QoS degradations. However, this communication should be performed without (or only slight) code modification of the available services; furthermore, the extension of the set of available services with a new service should be easy-to-do. This means that a well-defined QoS manager-service interface should be provided; the primitives the interface provide should be similar for all available (and future) services. Obviously, the implementations of these primitives will depend on the service in question.

3.4 Processing protocol data unit (PPDU)

The processing protocol data unit allows to transform (1) messages received from the QoS manager and the user interface into messages that implements CQoSM (see Section 4), and (2) messages received from neighbouring agents (for a given multicast tree) into messages which can be processed (understandable) by the internal components. Example of these transformations follow: when the user selects a service via the user interface, the latter sends a notification to PPDU; then, PPDU builds Ask_QoS_Info() (see Section 4) and sends it to its upstream neighbouring agent; upon receipt of Give_QoS_Info() signal, PPDU sends the information (about available QoS) to the user interface to be displayed to the user.

4 PROTOCOLS FOR INTER-AGENT COMMUNICATIONS

In this section we present a description of the operations of an agent that implements CQOSM; the operations described below are applicable for a single multicast tree; this means that the agent should perform these operations for each multicast tree that uses the component that hosts the agent.

Signals description

We define the following signals:

- Ask_QoS_Info (Tree_Id, Sender_Id, Receiver_Id): It is sent by the agent, identified by Sender_Id, to its upstream neighbouring agent identified, by Receiver_Id; Tree_Id indicates the identifier of the multicast tree in question.
- Give_QoS_Info (Tree_Id, Sender_Id, Receiver_Id, List_QoS): Its is sent by the agent, identified by Sender_Id, to its downstream neighbouring agent, identified by Receiver_Id; List_QoS is a list of QoS classes that are available from the agent identified by Sender_Id.
- Add_QoS (Tree_Id, Sender_Id, Receiver_Id, QoS): It is sent by the QoS agent, identified by Sender_Id, to its upstream neighbouring agent, identified by Receiver_Id; QoS indicates the QoS that the agent, identified by Sender_Id, wants to receive from the agent, identified by Receiver_Id.
- Remove_QoS (Tree_Id, Sender_Id, Receiver_Id, QoS): It is sent by the agent, identified by Sender_Id, to its upstream neighbouring agent, identified by Receiver_Id; QoS indicates the QoS that the agent, identified by Sender_Id, does not want to receive anymore from the agent, identified by Receiver_Id.
- Persuade (Tree_Id, Sender_Id, Receiver_Id, List_QoS): It is sent by the agent, identified by Sender_Id, to its downstream neighbouring agent, identified by Receiver_Id; List_QoS indicates the list of QoS classes that the agent, identified by Sender_Id, wants to deliver to the agent, identified by Receiver_Id.
- Viol (Tree_Id, Sender_Id, Receiver_Id, List_QoS): It is sent by the agent, identified by Sender_Id, to its upstream neighbouring agent, identified by Receiver_Id; List_QoS indicates the initially negotiated list of QoS classes that have been violated.
- Solve (Tree_Id, Sender_Id, Receiver_Id): It is sent by the agent, identified by Sender_Id, to its upstream neighbouring agent, identified by Receiver_Id;
- Available_QoS (Tree_Id, Sender_Id, Received_Id, List_QoS): It is sent by the

QoS agent, identified by Sender_Id, to its downstream neighbouring agent, identified by Receiver_Id; List_QoS indicates a list of QoS classes which are available from the agent, identified by Sender_Id.

Description of the operation of an agent

The operation of any type of agent is described in the following; obviously, the agents do not perform similar operations. To distinguish the operations of different agents, we have to remember that a user agent has no downstream neighbouring agents and a media source agent has no upstream agent.

To support adaptation of QoS, we assume that some internal monitoring mechanisms are available, which can detect violations of QoS provided by a given component. It is worth noting that facilities for monitoring will likely become *available with certain types of equipment* [10].

Variables description

We define the following variables:

- V_List_QoS, V_List_QoS' are variables which indicate lists of QoS classes;
- V_List_Agent is a variable which indicates a list of tuples (x,y) where x indicates an agent identifier and y a list of QoS classes;
- *Self* is variable which indicates the identifier of the agent in question;
- T indicates a time variable;
- V_Agent, V_agent1 are variables which indicate agent identifiers;
- Q, Q1 indicate QoS variables;
- V_Response is a variable which indicates a list of tuples (x,y), where x indicates a tree identifier and y an agent identifier. V_Response is a local variable at the agent level; initially V_Response=[] (this means that initially V_Response is empty);
- V_List_QoS1 is a variable which indicates a list of QoS classes; it is a local variable at the agent level; initially V_List_QoS1=[];
- V_Agents is a variable which indicates a list of tuple (x,y) where x indicates an agent identifier and y a QoS class;

Operation

- When a QoS violation is detected (let us assume that V_List_QoS indicates the agreed list of QoS classes which have been violated):

If the component that hosts the agent is not the cause of the violation, then

- the agent sends a Viol (Tree_Id, *Self*, *V_Agent*, V_List_QoS) where *V_Agent* is its upstream neighbouring agent (equal to the agent identifier in Upstream_Agent that corresponds to the entry Tree_Id in Tree_List in case of a router agent or in Si_Tree_List in case of media sink agent);
- V_List_QoS1=V_List_QoS; /* V_List_QoS1 is a local variable which will be used by the agent when Solve() signal is received; see below */

Otherwise, the agent performs the following operations:

- if the agent is a media sink agent /* this means that V_List_QoS consists of a single element which is available to the user*/, then
  - it sends Remove_QoS (Tree_Id, *Self*, *V_Agent*, V_List_QoS), where *V_Agent* is its upstream neighbouring agent; in this case V_List_QoS consists of a single element;

- it initiates a renegotiation with the user (via the user interface) to decrease the QoS currently provided;

Otherwise, the agent performs:

- C_List_QoS=C_List_QoS-V_List_QoS;
- for each tuple (*V_Agent*,) ∈ Downstream_Agents that corresponds to the entry Tree_Id (in Tree_List in case of a router agent or in So_Tree_List in case of a media source agent), the agent sends Available_QoS (Tree_Id, *self*, *V_Agent*, C_List_QoS);

endif

endif

- When a recovery is detected:

An agent that initially issued a Available_QoS() signal because of a local QoS violation, may monitor the current load of the component to check its capability to support a super set, V_List_QoS, of the currently provided QoS classes, C_List_QoS (ideally, V_List_QoS contains all QoS classes initially agreed). Upon the detection of such a capability, for each tuple (*V_Agent*,) ∈ Downstream_Agents that corresponds to the entry Tree_Id in Tree_List in case of a router agent or in So_Tree_List in case of a media source agent, the agent sends Available_QoS(Tree_Id, *Self*, *V_Agent*, V_List_QoS) signal.

- When Ask_QoS_Info (Tree_Id, Sender_Id, Receiver_Id) signal is received:

The agent (identified by Receiver_Id) sets V_List_QoS to A_List_QOS that corresponds to the entry Tree_Id in Tree_List in case of a router agent or in So_Tree_List in case of a media source agent. Then, the agent sends Give_QoS_Info (Tree_Id, *Self*=Receiver_Id, Sender_Id, V_List_QoS).

- When Add_QoS (Tree_Id, Sender_Id, Receiver_Id, *QoS*) signal is received:

The agent (identified by Receiver_Id) checks whether *QoS* ∈ C_List_QoS (A_List_QoS in case of a media source agent) that corresponds to the entry Tree_Id in Tree_List or in So_Tree_List.

If the response is yes, then the agent performs the following:

- it sends Persuade (Tree_Id, *self*, Sender_Id, V_List_QoS); the computation of V_List_QoS depends on the persuasion policies in use;
- it selects (depending on the persuasion policies in use; see below) a subset, V_Agents, of Downstream_Agents that corresponds to the entry Tree_Id in Tree_List or So_Tree_List; for each (*V_Agent*,) ∈ V_Agents, the agent builds and sends Persuade (Tree_Id, *self*, *V_Agent*, V_List_QoS); The computation of V_List_QoS depends on the persuasion policies in use;

Otherwise,

if the agent is a router agent, then it sends Add_QoS (Tree_Id, *self*, *V_Agent*, *QoS*), where *V_Agent* indicates its upstream neighbouring agent, and it updates its V_Response (V_Response=V_Response ∪ [(Tree_Id, Sender_Id]);

endif

- When a Remove_QoS (Tree_Id, Sender_Id, Receiver_Id, *QoS*) signal is received:
The agent (identified by Receiver_Id) updates the attributes of the entry Tree_Id in Tree_List or in So_Tree_List:
  - Downstream_Agents=Downstream_agents - [(Sender_Id, *QoS*)]
  - if not ($\exists$ *V_Agent*) such that [(*V_Agent*, *QoS*)] $\subseteq$ Downstream_Agents then
    - C_List_QoS=C_List_QoS-[*QoS*];
    - the agent sends Remove_QoS(Tree_Id, *self*, *V_Agent1*, *QoS*) where *V_Agent1* is its upstream neighbouring agent;
  endif
  - the agent selects (depending on the persuasion policies in use) a subset, V_Agents, of Downstream_Agents that corresponds to the entry Tree_Id in Tree_List or So_Tree_List; for each (*V_Agent*,) $\in$ V_Agents, the agent builds and sends Persuade (Tree_Id, *self*, *V_Agent*, V_List_QoS); The computation of V_List_QoS depends on the persuasion policies in use;

- When a Persuade (Tree_Id, Sender_Id, Receiver_Id, List_QoS) signal is received:
If the local variable V_Response is empty (which means that the agent is a sink agent and List_QoS is a single QoS; List_QoS=[*QoS*]), then the agent presents to the user, via the user interface, the QoS (*QoS*) which can be provided to him/her; if the user does not accept this QoS, the agent sends Remove(Tree_Id, *self*, Sender_Id, *QoS*).
Otherwise /*V_Response $\neq$ [] */, the agent performs the following:
  - C_List_QoS=List_QoS;
  - finds *V_Agent*, such that [(Tree_Id, *V_Agent*)] $\subseteq$ V_Response, and sends Persuade(Tree_Id, *self*, *V_Agent*, C_List_QoS);
endif

- When a Viol (Tree_Id, Sender_Id, Receiver_Id, List_QoS) signal is received:
The agent performs the following operations:
  - V_List_Agent=[(Sender_Id, List_QoS)];
  - it initiates and starts a timer, *Timer*; the value of the timer is computed based on statistics gathered during past behaviors of the system;
  - T=Current_Time; /* the agent reads the current time, Current_Time from a local clock */
  - while (Current_time<T+*Timer*) do
    if a Viol (Tree_id, *V_Agent*, Received_Id, V_List_QoS) signal is received, then Add(V_List_Agent, (*V_Agent*, V_List_QoS)); /* Add(l,x): adds x to the end of the list l */
    endif
  endwhile
  - if for each tuple (*V_Agent*,) $\in$ Downstream_Agents that corresponds to the entry Tree_Id in Tree_List or in So_Tree_List, (*V_Agent*,) $\in$ V_List_Agent, then
    if the agent is router agent then

- the agent sends a Viol (Tree_Id, *self*, *V_Agent1*, V_List_QoS') signal, where *V_Agent1* corresponds to its upstream neighbouring agent, and V_List_QoS'= ∪ V_List_QoS for (, V_List_QoS) ∈ V_List_Agent;

else (the agent is a media source agent) /* this means that the media source machine has problems to deliver data with appropriate QoS, e.g. because of resource shortage */

- it computes the list, V_List_QoS, of QoS classes, it is able to currently provide, using some resource reservation protocols;
- A_List_QoS=V_List_QoS;
- C_List_QoS=V_List_QoS;
- for each tuple (*V_Agent*,) ∈ Downstream_Agents that corresponds to the entry Tree_Id, the agent sends Available_QoS (Tree_Id, *self*, *V_Agent*, V_List_QoS);

endif

else

for (*V_Agent*,) ∈ V_List_Agent, the agent sends Solve(Tree_Id, *self*, *V_Agent*);

endif

- When a Solve (Tree_Id, Sender_Id, Receiver_Id) signal is received:

The agent asks the routing protocol to find a new path between Sender_Id, and Receiver_Id that might support the QoS classes contained in V_List_QoS1. If the response is yes, a transition of traffic transmission from the old path to the new path is performed; otherwise, the agent performs the following operations:

if the agent is a router agent, then

- C_List_QoS=C_List_QoS-V_List_QoS1;
- for each tuple (*V_Agent*,) ∈ Downstream_Agent that corresponds to the entry Tree_Id, the agent sends Available_QoS (Tree_Id, *self*, *V_Agent*, C_List_QoS);

otherwise /*the agent is a media sink agent */

- it initiates a renegotiation with the user (via the user interface) to decrease the QoS currently provided;

endif

- When a Available_QoS (Tree_Id, Sender_Id, Received_Id, List_QoS) signal is received:

If the agent is a router agent, then

- if (*QoS* ∈ C_List_QoS) then C_List_QoS=C_List_QoS-[*QoS*]; /* this means that Available_QoS() is received because of some QoS violation */
- if (*QoS* ∉ C_List_QoS) then C_List_QoS=C_List_QoS ∪ [*QoS*]; /* this means that Available_QoS() is received because of some recovery */
- for each tuple (*V_Agent*,) ∈ Downstream_Agents that corresponds to

the entry Tree_Id, the agent sends Available_QoS (Tree_Id, *self*, *V_Agent*, C_List_QoS);
otherwise /*the agent is a media sink agent */
it initiates a renegotiation with the user (via the user interface) to increase or decrease the QoS currently provided;
endif

Let us note that the agents are responsible for updating their state variables when executing the protocols described above.

Persuasion policies

The persuasion idea is better explained by an example. Let us assume that at a certain time a router agent, a, (part of Tree_Id) is delivering QoS_1 to a_1, a_2, et a_3, QoS_2 to a_4 and a_5, where $a_1, ..., a_5$ are the agent's downstream agents. In the following we present simple persuasion cases:

(1) if a_5 sends Move_QoS(Tree_Id, a_5, a,QoS_2), then the agent sends persuade (Tree_Id, a, a_4, QoS_1). This will allow (1) the agent to handle only QoS_1: the component (where the agent is installed) resources already reserved to support QoS_2 will be de-allocated and may be used to support new sessions; (2) the system to de-allocate the network resources used to deliver (from its upstream agent) QoS_2 to a. Furthermore, the same scenario may be executed by its upstream agent, ua; this depends on the state of ua.

(2) if a_6 and a_7 send Add_QoS(Tree_Id, a_6, a, QoS_1) and Add_QoS(Tree_Id, a_7, a, QoS_1) respectively, then the agent sends Persuade(Tree_Id, a, a_4, QoS_1) and Persuade(Tree_Id, a, a_5, QoS_1). This operation has similar effect as (1).

It is obvious that the policy used in this example depends mainly on the number of downstream agents asking for specific QoS classes; when the number of agents asking for QoS_1 is higher than the number of agents asking for QoS_2, then the agent persuades the agents to receive only QoS_1. The policy presented here is a simple one; however, more sophisticated ones may be used. These can be based on some complex optimizations procedures to increase the system benefits without discouraging clients. This means the persuasion of users will be based on some cost incentives. Another policy, may compute an average of QoS delivered and persuade all downstream agents to receive this average. We are still working on the specification and evaluation of different policies to be used in our CQoSM.

5 CONCLUSION AND OUTLOOK

In this paper, we described a multi-agent architecture for our new Cooperative Quality of Service Management. The basic idea is that agents installed on every node of the distributed system cooperate with each other in order to provide the QoS requested by the different participants of the application.

We are currently in the process of implementing a sample application based on this architectural framework, namely a tele-teaching application where a lecture given by a teacher can be "virtually" attended by a large number of students using their own workstation. The application is not developed from scratch; rather, it is based on existing code from our previous news-on-demand prototype and on the MBone Tools *vic*, *vat*, *wb* [8] and *vcr* ([7], to record a session). Contrary to "normal" MBone applications, we only handle one media stream per group address and with one tool instance. This

approach allows us to switch between qualities simply by stopping the currently running instance of the tool and starting a new one with a new multicast address, resulting in the reception of the media stream in a different quality. For the agents, we are looking into possibilities offered by the Web languages Java and Perl which offer powerful constructs to handle distributed and cooperative environments. We are especially interested in agent mobility in order to provide for a more flexible agent distribution throughout the network.

6 ACKNOWLEDEMENT

This work was partially supported by a grant from the Canadian Institute for Telecommunication Research (CITR), under the Networks of Centres of Excellence Program of the Canadian Government.

7 REFERENCES

[1] C. Aurrecoechea, A. Campbell, and L. Hauw. *A Survey of QOS Architectures.* Multimedia Systems Journal, Special Issue on QoS Architectures, 1997. To appear.

[2] G. Dermler, T. Gutekunst, B. Plattner, E. Ostrowski, F. Ruge, and M. Weber. *Constructing a Distributed Multimedia Joint Viewing and Tele-Operation Service for Heterogeneous Workstation Environments.* In Proceedings of the IEEE Workshop on Future Trends of Distributed Computing, Lisbon, Portugal. IEEE Computer Society Press, 1993.

[3] S. Fischer, A. Hafid, G. v. Bochmann, and H. de Meer. *Cooperative Qos Management in Multimedia Applications.* In N. Georganas, editor, IEEE International Conference on Multimedia Computing and Systems (ICMCS'97), Ottawa, Canada. IEEE Computer Society Press, June 1997. To appear.

[4] S. Fischer and R. Keller. *Quality of Service Mapping in Distributed Multimedia Systems.* In Proceedings of the IEEE International Conference on Multimedia Networking (MmNet95), Aizu-Wakamatsu, Japan, pages 132–141, September 1995.

[5] A. Hafid and G. v. Bochmann. *Quality of Service Negotiation in News-on-Demand Systems: An Implementation.* In A. Azcorra, T. D. Miguel, E. Pastor, and E. Vazquez, editors, Proceedings of the Third International Workshop on Protocols for Multimedia Systems, Madrid, Spain, pages 221–240, Oct. 1996.

[6] A. Hafid and G. v. Bochmann. *Quality of Service Adaptation in Distributed Multimedia Applications.* ACM Multimedia Systems Journal, 1997. To appear.

[7] W. Holfelder. *MBONE VCR – Video Conference Recording on the MBONE.* In P. Zellweger, editor, ACM Multimedia '95 (Proceedings), pages 237–238, New York, Nov. 1995.

[8] V. Kumar. *MBone – Interactive Multimedia on the Internet.* New Riders Publishing, Indianapolis, Indiana, 1996.

[9] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A transort protocol for real-time applications.* RFC-1889, Internet Engineering Task Force, Audio-Video Transport Working Group, 1996.

[10] A. Seneviratne and H. S. Cho. *Quality of Service Mapping in Distributed Multimedia Systems.* In Proceedings of the IEEE International Conference on Multimedia Networking (MmNet95), Aizu-Wakamatsu, Japan, pages 126–131, September 1995.

[11] D. Seret and J. Jung. *Translation of QoS Parameters into ATM Performance Requirements in B-ISDN.* In IEEE Infocom'93, San Francisco, 1993.

[12] R. Somalingam. *Network Performance Monitoring for Multimedia Networks.* Master's thesis, McGill University, Montreal, Canada, 1996.

8 BIOGRAPHY

Dr. Abdelhakim Hafid is a Researcher Staff Member at the Computer Research Institute of Montreal (CRIM) working in the area of distributed multimedia applications; he is also an Adjunct Professor at University of Montreal, Department of computer Science and Operationel Research. He received his Masters and Ph.D. degrees in computer science from University of Montreal on quality of service management for distributed multimedia applications in 1993 and 1996, respectively. From 1993 to 1994 he was visiting scientist at GMD-FOKUS, Systems Engineering and Methods group, Berlin, Germany working in the area of high speed protocols testing. His current research interests are in broadband multimedia services and communications.

Dr. Stefan Fischer is currently a postdoctoral researcher at the University of Montreal. He got his diploma in Computer Science applied to Business Administration and his doctoral degree in Computer Science from the University of Mannheim, Germany, in 1992 and 1996, respectively. His research interests include distributed multimedia systems, quality of service and formal methods for high-speed networks and applications. He is also the author of two books on network programming and on intranets.