[7] 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.

[8] A. Hafid and G. v. Bochmann. Quality of Service Adaptation in Distributed Multimedia Applications. ACM Multimedia Systems Journal, 1997 (to appear).

[9] 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.

[10] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Transactions on Networking*, 1(3):286–292, June 1993.

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

[12] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *ACM SIGCOMM'96*, Stanford, CA, Aug. 1996.

[13] 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.

[14] 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.

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

[16] N. Shacham. Multipoint communication by hierarchically encoded data. In *IEEE Infocom'92*, pages 2107–2114, 1992.

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

[18] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 7(5), Sept. 1993.

7. 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 QoS 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 teleteaching 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* [11] and *vcr* ([9], 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.

Acknowledgements

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.

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] A. Ballardie, J. Crowcroft, and P. Francis. Core based trees (CBT) – An Architecture for Scalable Inter-Domain Multicast Routing. In *ACM SIGCOMM '93*, pages 85–95, 1993.

[3] S. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, 1991.

[4] 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.

[5] 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.

[6] 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.

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,*) $\in$ 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 ($QoS \in$ C_List_QoS) then C_List_QoS=C_List_QoS-[QoS]; /* this means that Available_QoS() is received because of some QoS violation */
- if ($QoS \notin$ C_List_QoS) then C_List_QoS=C_List_QoS $\cup$ [QoS]; /* this means that Available_QoS() is received because of some recovery */
- for each tuple (*V_Agent,*) $\in$ 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

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 $a_1, \ldots, a_5$ 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.

If the local variable V_Response is empty (which means that the agent is a sink agent and List_QoS is a single 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'= $\cup$ V_List_QoS for (, V_List_QoS) $\in$ 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, e.g. RSVP [12];
- A_List_QoS=V_List_QoS;
- C_List_QoS=V_List_QoS;
- for each tuple (*V_Agent*,) $\in$ 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*,) $\in$ 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

agent or in So_Tree_List in case of 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 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* $\in$ 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, [*QoS*]);
- 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*,) $\in$ 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;

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 $\cup$ [(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:

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 indicate agent identifier variables;
- 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:

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) and V_List_QoS indicates the agreed list of QoS classes which have been violated;
- 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 Move_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 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

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 a 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* [14].

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

ping 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 [17].

4.3. 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 [13] (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.

4.4. 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 5), and (2) messages received from neighbouring agents (for a given multicast tree) into messages which can be processed (understandable) by the internal components. Examples of these transformations are:

- When the user selects a service via the user interface, the latter sends a notification to PPDU. Then, PPDU builds Ask_QoS_Info() 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.

- When the QoS manager performs local QoS negotiation with success (after the user specifies his/her QoS requirements), it sends a notification to PPDU. Then, PPDU checks whether the requested QoS is provided by its upstream agent (by using the information received in Give_QoS_Info()). If the response is yes, PPDU builds an add() signal and sends it to its upstream neighbouring agent; otherwise it sends a notification to the user (via the user interface) for renegotiation.

-When the QoS manager detects a QoS violation, it sends notification to PPDU. Then, PPDU builds a viol() signal and sends it to its upstream neighbouring agent.

- When PPDU receives a persuade() signal, it communicates this information to the user interface.

5. 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

4.1. 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. This interface should hide, as much as possible, the internal QoS parameters, e.g. throughput and jitter, and provide facilities to describe the requested QoS in terms of a set of user-perceived characteristics of the performance of a service. The user preferences are described in terms of (1) QoS setting for video, audio, still images and text and, (2) the cost he/she is willing to pay to play the requested document with the desired quality. QoS interface also provides facilities to set not only the *desired QoS* but also the *worst acceptable QoS* values. The user specify his/her requirements from each participant of the session; when a user wants to join a session, the interface displays the list of the current participants. The user has the choice: (1) to ask for similar QoS requirements from all the participants; he/she has to select all participants and then specify single QoS; or (2) to ask for a specific QoS from each participant; he/she has to select participants individually (or by groups) and specify single QoS; 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). We believe that the second choice should be provided by such an interface, since QoS requirements of a user might be different depending on the participants in question; in a teleteaching application, a student will likely require higher QoS from the lecturer than from a participating student. A detailed description of QoS interface in the context of remote access to MM database can be found in [7]; the main parts of this interface 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 [9]. 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. The activation of these operations depends on the semantics of the application; in a teleteaching application, only the lecturer can invite or quick a participant.

4.2. 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 (2) analytical functions which is difficult to realize (an example of AAL-ATM mapping QoS parameters can be found in [15], and examples for user-transport mapping are given in [6]); or (2) map-

3.3. 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 7).

The main role of a media source agent is to ask (when appropriate) the source to transmit information with 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 protocol 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 7. State variable of a media source agent

4. Media sink agent

The media sink agent is one the main component of the architecture of CQOSM. Figure 8 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 session control functions is out of scope of the paper and can be found elsewhere [4].

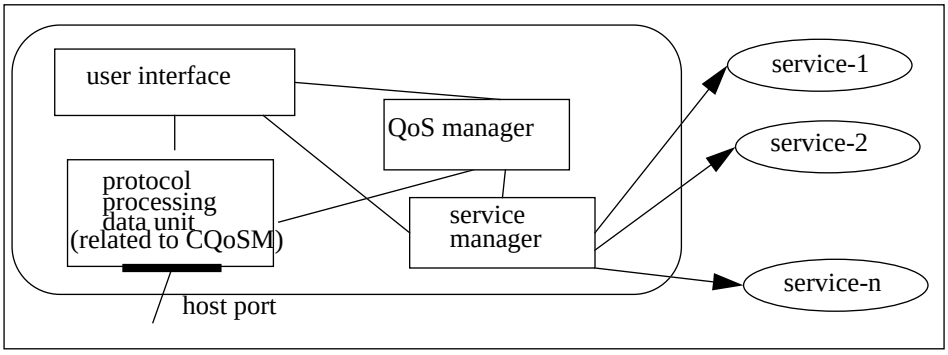

Figure 8. Media sink agent architecture

3.2. 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 (Figure 5). A media sink agent provides means to the user, via a user interface: (1) to specify 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) to specify the maximum cost he/she willing to pay to be a participant in a session. This operation is performed each time the user asks for a MM service. Examples of these services are audio-conferencing, video-conferencing, and sharing MM applications.

The user does not have to use all the services at the same time; depending on his/her needs, the user will use one or more services over the session duration. For example, at the beginning of a cooperative session, a user uses an audio-conferencing service, e.g. *vat* [9], and video-conferencing service, e.g. *vic* [9], to communicate with other participants; a couple of minutes later, he/she decides to use a MM application sharing service, e.g. JVTOS [4], to share a MM document with others.

Si_Tree_List

Tree_Ident	Upstrean_agent (U_Agent_Id)
⋮	⋮

Figure 5. State variable of a media sink agent

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 (Figure 6). 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.

The number or the characteristics of MM services provided at the host machine should not affect the implementation of the user agent; this means that the agent should be able to communicate, e.g. start and stop primitives, with all (present and future) MM services. For this reason, an abstract interface which is common for all services is defined; obviously the implementations of the primitives provided by this interface are customized for each MM service; however, this customization is transparent to the agent. To add a new MM service, for instance, one has only to implement the interface without code modification of the agent.

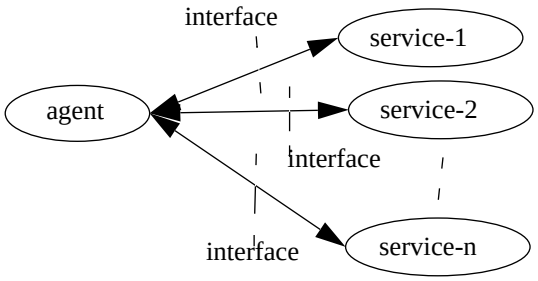

Figure 6. Sink media agent-MM services interactions

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.

3.1. Router agent

A router agent is located in any router of the system; to support CQoSM, it maintains a state variable which we call Tree_List (Figure 3). 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. Obviously, C_List_QoS is a sub-list of A_List_QoS. 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 3. 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 (Figure 4). Obviously, the implementation of such an interface will depend on the routing protocol in use; however, this will be transparent for router agents.

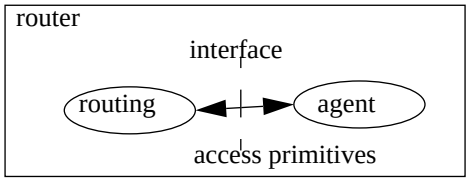

Figure 4. Router access primitives

router could be freed and the communication service cost would be much lower for receiver r1 which would be the motivation for him to switch. Assume that he pays 10 money units for the high-quality stream. If the system is able to offer him the low-quality stream, which is black&white instead of colored, for 1 money unit, it would probably be very tempting to switch.

If r1 finally switches, a new situation for the other agents occurs. Consider agent a1 now. It realizes that only the subtree of agent a3 receives the high-quality video. It may now try to persuade a3 to switch the quality which in turn could lead to a3 sending a switch proposal to r5.

3. A Multi-agent architecture for cooperative QoS management (CQoSM)

Figure 2 presents an architecture of the cooperative QoS management 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 2 is essentially independent from the type of applications, e.g. tele-medicine or tele-teaching, 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.

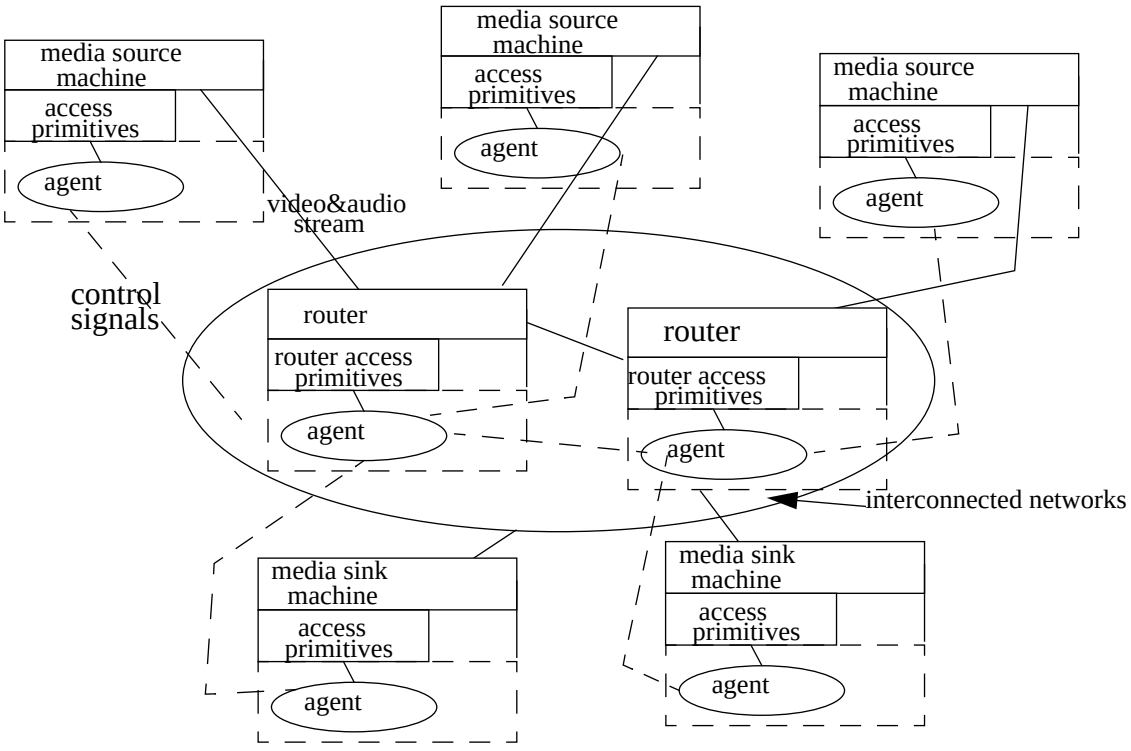

Figure 2.A multi-agent architecture for CQoSM

The system that supports QoS management for MM applications can be easily extended with new components without code modification of the existing agents; one has only to implement the

niques are independent of underlying protocols and mechanisms and work for different coding and routing techniques, such as hierarchical video encoding [16], multicast routing already including resource reservations as discussed e.g. in [10], traditional MBone routing techniques [3] or video selection using group management protocols [12]. Our initial considerations were based on the multicast routing protocol core-based tree routing (CBT) [2][1] and the resource reservation protocol RSVP [18].

A QoS agent provides the following QoS functions:

- QoS negotiation
  It occurs when a new user requests to become part of the application and receive some of its streams. The multicast routing scheme will forward its request until it arrives at a router that is already participating in the requested application, i.e., supporting its multicast tree. The agent of this router then contacts the new user's agent and sends the information about all available streams (quality and cost). Note that there is no central instance providing quality and cost information, since the cost for available qualities may differ significantly from one region to another. The user may select the streams he/she desires. Connections are set up by the underlying protocols; the QoS agents update their information about supported streams. We developed protocols to fulfil these tasks [5].
- QoS adaptation
  This functions becomes active when a component is no longer able to support the currently negotiated QoS, e.g. due to overload, failure or other stochastic situations. The QoS management system then tries to find a way to continue providing the service, either by selecting another component or by lowering the service quality within the boundaries negotiated with the client. In the framework of Cooperative QoS Management, we developed a protocol between QoS agents that helps to detect QoS violations and locate their source; furthermore, QoS agents can initiate the adaptation process by several means, one of them being to request the partial reconfiguration of the multicast tree in the area where the problem occurred. More details on the adaptation protocols can be found in [5].
- QoS renegotiation
  Traditionally, there are two types of QoS renegotiation, namely system-initiated and user-initiated. The former occurs when a negotiated QoS was violated and the QoS adaptation was not able to fix the problem. Then the system proposes the user to negotiate a lower quality. The latter happens when users are no longer content with the quality they negotiated. In such a case, they start a new negotiation process to switch to another quality.
  Within Cooperative QoS Management, system-initiated renegotiation may also occur when the management detects an unsatisfying situation concerning resource usage as described in Section 1. Consider again the example in Figure 1. Receiver r1 is the only one on its subtree that receives the high-quality video, i.e., he is the exclusive user of the resources reserved for this stream. QoS agent a3 realizes this, and after checking several other parameters, it decides to propose to r1 to switch to the lower quality. If it already has the necessary information, r1's agent may take the decision on its own, but it may also contact the user and ask if he would like to switch. Certainly, users may forbid their agents to forward any such requests to them, in order to not be disturbed in their session.
  If r1 considers to switch to the lower quality, resources for the high-quality video on a3's

1. In Core-Based Tree routing, there is only one multicast tree per receiver group. All streams are first unicast to the root of this tree (the core) and from there multicast to the receivers. Several optimizations are possible.

2. Overview of cooperative QoS management

Cooperative QoS management has been developed with multimedia applications in mind, in which many users participate at the same time, such as tele-education systems or life video transmissions of major sports events. We assume that single data streams are multicast to many users and that senders offer the same media stream in several qualities, e.g. a high, a medium and a low quality video stream. There are no individual QoS negotiations between senders and receivers; rather, receivers have to select among the qualities offered by the senders.

The basic idea of our new scheme is to install an application-oriented QoS agent on each router of the underlying network and on every end system participating in an application. These QoS agents are able to communicate with their neighbouring agents, informing them e.g. about current QoS values supported in their local area or about possible QoS problems. This knowledge is basically *application-oriented*, i.e. the agents know about QoS requirements and negotiated values for users as well as relationships between streams. This constitutes a main difference of our approach compared to existing QoS management functions on network nodes which deal with lower-layer QoS, such as ATM cell loss priority etc., and which do not have any information about relationships between streams and applications.

In our approach, however, not every agent may contact any other agent. Rather, communication depends on the existing multicast trees, leading to a hierarchical communication structure. For each multicast tree in which a given router is involved, the QoS agent knows its upstream and all downstream neighbours. If the neighbouring node is an end system, the agent knows all receivers on this end system. A receiver's QoS agent knows only its upstream QoS agent(s), and a sender's agent its downstream neighbours. The information about neighbours may be easily set up during the establishment of the multicast tree, resp. when a member leaves or a new member joins.

As an example, consider the situation displayed in Figure 1 where one sender is multicasting one high-quality video (the regular arrows) and one low-quality video (the dashed arrows) to a group of receivers.

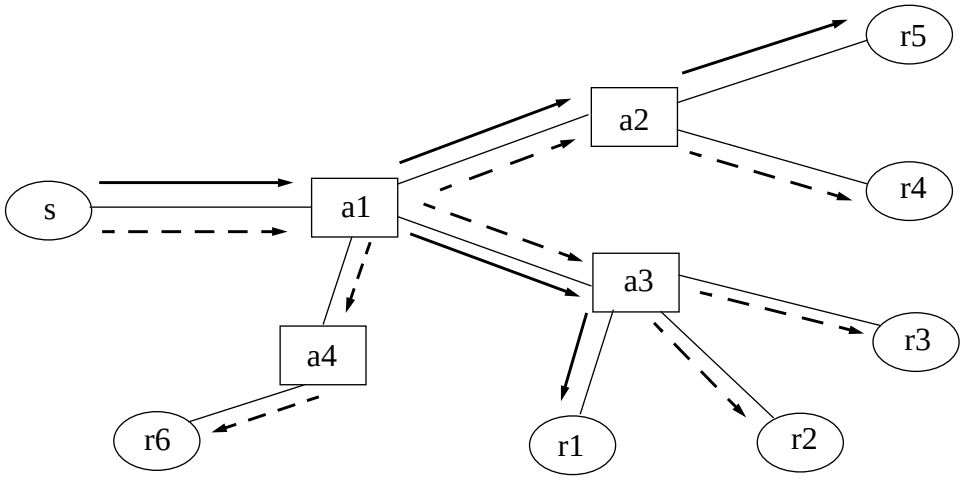

Figure 1. Multicasting streams of different qualities

Every router in the network has to forward all the streams which are requested by users connected via this router. The QoS agent a2 knows its downstream agents r4 and r5 as well as its upstream agent a1. It also has information about the available resources on its router and the cost associated with reserving them. Finally, it knows all streams available for this application and has access to the multicast routing and resource reservation protocol running on this router. Note that our tech-

for all participating users, since some users may participate with a very limited local workstation which cannot provide the quality which is adopted by the majority of the conference participants. We therefore adopt the premise that different levels of quality, often corresponding to different levels of cost, must be provided in the context of distributed multimedia applications.

Much work on 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 have 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 [8], 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 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 [7]. 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, as e.g. for remote database access of single users, is not workable any more, because the number of users involved is too large 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* [5] (*CQoSM*), where so-called *QoS agents* are installed on the routers and end systems participating in an applications. These agents cooperate with each other in order to provide the QoS levels requested by the application. An interesting new feature, compared to other QoS management schemes, which becomes possible due to this decentralized approach, is the possibility of communication between users resp. their local QoS agents, allowing for a cooperative selection of desired qualities. 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.

In this paper, we present a multi-agent architecture for CQoSM which will later serve as a framework for implementations. Therefore, we first give a brief introduction into CQoSM in Section 2. In Section 3, we describe in detail the multi-agent architecture in terms of media source, media sink and router agents. Section 4 further details on one of these agents, namely the media sink agent. In Section 5, we present the protocols which are executed between the agents in order to provide a cooperative solution for all arising QoS problems. Sections 6 finally concludes the paper and gives an outlook on future work such as a prototype implementation.

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 design of distributed multimedia applications requires careful consideration of quality of service (QoS) issues, because the presentation quality of live media, especially video, requires relatively high utilisation of networking bandwidth and processing power in the end systems. Much work on QoS management has been done in the context of applications involving two (or at most a few) communicating entities, such as access to remote multimedia databases by single users. Thus, the approaches developed so far are not workable 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

Quality of Service (QoS) has become a major issue in today's high-speed distributed systems such as multimedia applications, because the presentation quality of live media, especially video, requires relatively high utilisation of networking bandwidth and processing power in the end systems. For applications running in a shared environment, the allocation and management of these resources is an important question, although most existing systems are based on a best-effort approach.

From a user's point of view, best-effort approaches are not suitable for distributed multimedia systems: some users may be ready to pay some higher price for obtaining a maximum quality, while others may prefer low-cost presentations with lower quality. As soon as a user pays for a certain service quality, the provider has to give *service quality guarantees*. In addition, for a teleconferencing application involving many users, a single quality of service level may not be appropriate