ELSEVIER Computer Networks and ISDN Systems 30 (1998) 777–794

COMPUTER NETWORKS and ISDN SYSTEMS

A quality of service negotiation approach with future reservations (NAFUR): a detailed study

Abdelhakim Hafid [*], Gregor von Bochmann [1], Rachida Dssouli [2]

Université de Montréal, Dept. d'IRO, Groupe de Téléinformatique, Montréal, Canada H3C 3J7

Abstract

Distributed multimedia (MM) applications such as video-on-demand and teleconferencing provide services with different quality of service (QoS) requirements. Hence, the user should be able to negotiate the desired QoS depending on his/her needs, the end-system characteristics and his/her financial capacity. In response to a service request with the desired QoS, most QoS negotiation approaches return an acceptance or a simple rejection of the request. More specifically, they provide the user only with the QoS that can be supported at the time the request is made and assume that the service is requested for indefinite duration. This paper describes work on a new QoS negotiation approach with future reservations (NAFUR) that decouples the starting time of the service from the time the service request is made and requires that the duration of the requested service must be specified. NAFUR allows to compute the QoS that can be supported for the time the service request is made, and at certain later times carefully chosen. As an example, if the requested QoS cannot be supported for the time the service request is made, the proposed approach allows to compute the earliest time, when the user can start the service with the desired QoS. NAFUR will help to increase (a) the flexibility of the system by providing the user with more choices, and (b) the system resource utilization, and the availability of the system, by encouraging the sharing of the resources, e.g. multicast for video-on-demand systems. Furthermore, it provides the flexibility to incorporate (a) a range of resource reservation schemes and scheduling policies, and (b) a range of new system component technologies. © 1998 Elsevier Science B.V.

Keywords: Quality of service; Negotiation; Distributed multimedia applications; Future reservation

1. Introduction

The new distributed multimedia (MM) applications are characterized by handling continuous media and by managing various media at the same time. Different types of continuous media require different levels of quality of service (QoS), and they require guarantees for the level of service to be maintained. This implies stringent requirements for the communication systems and the end-systems to support the requirements of MM applications. Hence, these applications need end-to-end QoS management, particularly QoS negotiation, to ensure that the requirements of the users are satisfied.

Most existing QoS negotiation protocols [1–8] are only concerned with the communication quality in terms of QoS parameters, such as throughput, delay and jitter. Furthermore the negotiation results, in response to the user request, are restricted to an acceptance or rejection of the request. This implies

[*] Corresponding author. E-mail: hakim@csd.uwo.ca.
[1] bochmann@iro.umontreal.ca.
[2] dssouli@iro.umontreal.ca.

that a second attempt of the user cannot take advantage of information obtained through the first request to change, if possible, the requirements to fit the current system load. In [9] a QoS negotiation protocol based on the Tenet protocol suite [10] has been proposed to improve the information given to the users by the network when a connection request is rejected. Such improvement is closely related to the characteristics of the Tenet protocol suite, e.g. admission control tests. Also application-to-application negotiation protocols [11–14] focus only on the establishment of an agreement between the parties with respect to the application QoS parameters. More generally, the service model of the existing negotiation approaches provides the user with the QoS that can be supported at *the time the service request is made*, and assumes that the service is requested for *indefinite duration*. We believe that such approaches do not fit the needs for future MM service providers and users. Let us present an example which motivates this claim.

We consider a video-on-demand application which supports remote access to MM databases (Fig. 1). In the following we present two basic situations of user–server interactions in the existing negotiation approaches.

(a) Let us assume that a user located at *client-1* asks to play a movie with a desired QoS, e.g. (video_color = color, video_rate = TV-rate, video_window_size = large), located at *server-1*. To support this service request *server-1, MM transport system* and *client-1* must commit to reserve resources to support a level of QoS, e.g. delay, in such way that the end-to-end QoS is satisfied [15]. Unfortunately, at the time the request is made there are not enough resources to support the service requested. Making use of the existing negotiation approaches a rejection will be sent to the user.

A desirable QoS negotiation approach will provide the user with two proposals: (1) the requested service can be provided immediately with a degraded QoS, e.g. (video_color = black and white, video_rate = TV-rate, video_window_size = large), and (2) the requested service can be provided with the desired QoS at a future time T_1, e.g. 30 minutes later. Thus, the user can choose between the two proposals depending on his/her requirements.

(b) Let us assume that the user selects the second proposal, and at a later time $T_2 < T_1$, a second user on *client-2* asks for the same movie. Fortunately, the system has enough resources to support the second user request at T_2 (immediately). Making use of the current negotiation approaches an acceptance will be sent to the user.

A desirable negotiation approach will check the possibility of supporting the second user request concurrently with the first user at time T_1, since this may lead to sharing of resources and possibly lower cost. If this is the case, two proposals will be sent to the second user: (1) an acceptance to start the movie immediately (at T_2) with a cost $cost_1$, and (2) an acceptance to start the movie later at T_1 with a cost $cost_2$ where $cost_2 < cost_1$. Consequently, the user can select the proposal which corresponds to his/her wishes and constraints. The computation of the second proposal is motivated by the fact that servicing users individually is inefficient, expensive and not scalable. One of the best ways to deliver information to more users is through the use of multicast communication which has the advantage of being scalable. We believe that a large number of users will select the second proposal which will optimize resources usage, e.g. multicast communication [16], and minimize cost for the users.

To increase the flexibility and the availability of future MM systems (a) the starting time of the requested service should be decoupled from the time the service request is made, and (b) the duration of the service requested should be specified by the service user. In the present paper we propose a QoS negotiation approach with future reservations (NAFUR) that supports these principles. To make use of NAFUR, the user must specify, besides the desired QoS, only the duration of the requested service.

There is a basic assumption in our approach: We assume that the system is built in the framework of QoS guarantees, that is, the components are able to reserve resources to support certain levels of QoS.

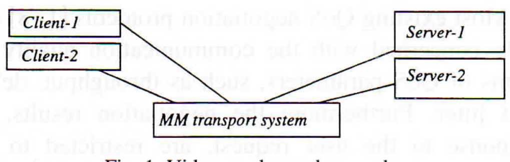


Fig. 1. Video-on-demand example.

Consequently, NAFUR provides the flexibility to incorporate a range of existing resource reservation schemes and scheduling policies, and a range of new system component technologies, such as ATM.

The paper is organized as follows. Section 2 describes the operation of NAFUR at the user interface; it also presents some performance analysis of NAFUR by means of simulations. Section 3 describes the operation of NAFUR in a hierarchical multi-domain environment; some ideas to optimize the operation of NAFUR are also discussed. Section 4 discusses the case of unknown services durations. Finally, Section 5 summarizes our results and presents some concluding remarks.

2. QoS negotiation interactions at the user interface for present and future service sessions

In this section we model a distributed system as a single module and a QoS manager (Fig. 2) which represents the access point to the module. The ability of the QoS manager to process a service request is directly related to the admission criteria which the QoS manager uses to decide whether a new request is accepted. This criteria is that the sum of previously assigned resources plus the resources required by the new request should not exceed the resources of the system. In the case of acceptance, the QoS manager reserves the required resources. When the QoS manager receives a request to terminate the service it simply de-allocates the reserved resources. When a renegotiation request is received, the QoS manager may decrease the amount of the reserved resources if the new QoS is less restrictive than the QoS currently provided, otherwise it checks the admission criteria with the new QoS constraint and the current load of the system. In this section, the terms

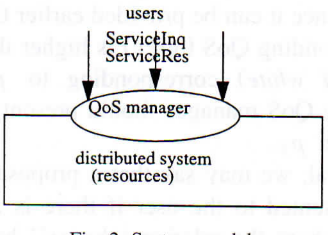


Fig. 2. System model.

"system" or "QoS manager" will be used interchangeably.

More specifically, the following operations are provided by the QoS manager to the users (Fig. 2):

1. ServiceInq (in *req*: Request; *resPeriod*: Time; out *pro*: Proposals);
2. ServiceRes (in *req*: Request; out *s*: Status);

The operation ServiceInq makes an inquiry about the availability of a particular service characterized by a request *req*. This parameter is a tuple of three elements $req = \langle Q, starttime, length \rangle$, where Q is a set of QoS parameters characterizing the quality of the requested service, *starttime* is the desired starting time for the service, and *length* is the length of the time for which the service is requested; the period for which the service is requested is therefore the time interval $[starttime, starttime + length]$. The result *pro* is a set of proposals where each proposal indicates the QoS available for a period of *length* at some future times. Formally, a proposal is defined as a tuple $\langle time, QoS \rangle$ where *QoS* represents the QoS that can be supported by the system over the interval $[time, time + length]$.

The parameter *resPeriod* (argument of ServiceInq) indicates for how long the service reservations (made to support *pro*) should be kept (on a temporary basis) until a subsequent invocation of the ServiceRes operation will make an effective reservation for a particular proposal. The operation ServiceRes is used to effectively make a service reservation. Typically, it will be called after the execution of a ServiceInq operation, and its *req* parameter will correspond to one of the proposals contained in *pro*. The status parameter, *s*, indicates the success or failure of the operation.

To support the operation ServiceInq, we assume that the QoS manager has enough knowledge about the available QoS, presently and in the future; this information is called available service projection (*asp*). Formally *asp* consists of a list of tuples (*time, QoS*) where *QoS* corresponds to the QoS that can be supported by the component at time *time*.

As mentioned above, the QoS manager should provide the user with the QoS that can be provided for the duration, *length*, of the service requested at certain times. However, *asp* contains only the QoS that can be supported by the system at a certain given time(s). Thus the QoS manager must be able to

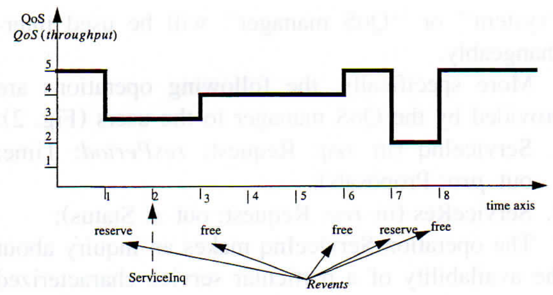


Fig. 3. An example of available service projection of the system.

transform *asp* to *pro*. This is the subject of Section 2.2.2.

2.1. Available service projection

The QoS manager contains the resource reservation state (of the system) which indicates the resource reservations made for all its resources at present and in the future. One way of representing this state is by a list, *REvents*, of reservation events which are sorted by the event time (Fig. 3). These events are of the following forms: ⟨*reserve, time, R, purpose*⟩ and ⟨*free, time, R, purpose*⟩, where *reserve* or *free* indicates the type of events, *time* is the event time, *R* is the amount of reserved resources, and *purpose* is a reference to the user (or the application) for whom the reservation was made. An event of type *reserve* (resp. *free*) means that an amount of resources *R* will be reserved (resp. de-allocated) for the user *purpose* at time *time*.

To produce the available service projection *asp*, the QoS manager makes use of some functions which allow to compute the available QoS given the available system resources (definitions of these functions is out of scope of the paper) and *REvents* (see example below). When ServiceRes (*req, s*) is invoked, the QoS manager generates a corresponding reserve event with an amount of resources corresponding to the QoS contained in *req* parameter.

2.1.1. Example

Let us consider a network with a maximum bandwidth, *R*, of 5 Mbits/sec. For sake of simplicity and clarity of the example, we do not specify how the shared bandwidth is controlled. Fig. 3 shows the list, *REvents*, of events that corresponds to the system, and the corresponding available service projection *asp*. *Revents* = [⟨*reserve*, 1, 2, ⟩, ⟨*free*, 3, 1, ⟩, ⟨*free*, 6, 1, ⟩, ⟨*reserve*, 7, 3, ⟩, ⟨*free*, 8, 3, ⟩]. For instance, ⟨*reserve*, 1, 2, ⟩ means that at time 1 a reservation of 2 Mbits/sec is made.

The QoS manager will produce the following: *asp* = [⟨2, $QoS_1 = 3$⟩, ⟨3, $QoS_2 = 4$⟩, ⟨6, $QoS_3 = 5$⟩, ⟨7, $QoS_4 = 2$⟩, ⟨8, $QoS_5 = 5$⟩].

In this example, the available QoS (throughput in Mbits/s) corresponds to the available system resources (bandwidth in Mbits/s).

2.2. Proposals

The QoS manager should provide the user with the QoS that can be provided for the duration, *length*, of the service requested starting at certain future times. However, *asp* (Section 2.1) contains only the QoS, *QoS*, that can be supported by the system at certain given points in time. Thus, the QoS manager must be able to transform *asp* to proposals.

2.2.1. Comparison of proposals

In response to a ServiceInq from the user, the QoS manager produces a list of proposals. However, not all the proposals are useful to be presented to the user. A kind of *filtering* must be supported by the QoS manager to compute only representative (''useful'') proposals.

For sake of clarity, let us consider the following example. In response to the user request to display a specific movie with a desired QoS (25 *frames/second, color*) the QoS manager produces (without the filtering facility) three proposals: p_1 = ⟨8*am*, (25 *frames/second, grey*)⟩, p_2 = ⟨8:20 *am*, (25 *frames/second, black and white*)⟩, and p_3 = ⟨8:30 *am*, (25 *frames/second, color*)⟩. It is obvious that the proposal p_2 will not be selected by the user; even if the user accepts a degraded QoS, he/she will select p_1 since it can be provided earlier than p_2 and the corresponding QoS (*grey*) is higher than the one (*black and white*) corresponding to p_2. Consequently, the QoS manager should present to the user only p_1 and p_3.

In general, we may say that a proposal *p* should not be presented to the user if there is a ''*better*'' proposal, where the relation ''*better*'' between two proposals $p_1 = \langle t_1, QoS_1 \rangle$ and $p_2 = \langle t_2, QoS_2 \rangle$ is defined as follows:

$$p_1 \; better \; p_2 \text{ if } t_1 \leq t_2 \text{ and } QoS_1 \geq QoS_2 .$$

Since QoS consists generally of a number of QoS parameters, the relation $\geq$ between two QoS, QoS_1 and QoS_2, must first be defined. One way to define the relation is by introducing weights which indicate the relative importance of the different parameters. We say that $QoS_1 \geq QoS_2$ if and only if the weighted average of the QoS parameter values corresponding to QoS_1 is higher than the one corresponding to QoS_2.

To compute the weighted average, we associate (1) a *numerical value* to each possible instance of a given QoS parameter, and (2) a *weight* to each QoS parameter. The *numerical values* should correspond to the quality of the different instances of a given QoS parameter, while the *weights* must indicate the importance of the different QoS parameters. For example, in a video-on-demand system [12], the parameters used to specify, at the user level, the quality of video are: color, frame rate, and display size; an instance of color can take the values *color*, *gray*, or *black and white*, an instance of frame rate can take *TV rate, reduced rate*, or *frozen rate*, while an instance of display size can take *large, medium*, or *small*; to each instance we associate a numerical value: ($color = 3$, $gray = 2$, $black\ and\ white = 1$), ($TV\ rate = 3$, $reduced\ rate = 2$, $frozen\ rate = 1$), and ($large = 3$, $medium = 2$, $small = 1$). Let us assume that $QoS_1 = (color, reduced\ rate, medium)$ and $QoS_2 = (gray, TV\ rate, medium)$. If we associate to color a weight of 0.75, to frame rate a weight of 0.125, and to display size a weight of 0.125, then $QoS_1 \geq QoS_2$. Indeed, the weighted average that corresponds to QoS_1 (resp. QoS_2) is equal to $3 \times 0.75 + 2 \times 0.125 + 2 \times 0.125 = 2.75$ (resp. 2.125). The determination of the weights depends on the nature of the service considered, e.g. delay sensitive services or reliability sensitive services.

Now that the relation *better* is defined, it is interesting to mention its properties.

- *better* is reflexive: p_1 *better* p_1 for any proposal $p_1 = \langle t_1, QoS_1 \rangle$ since $t_1 \leq t_1$ and $QoS_1 \geq QoS_1$;
- *better* is asymmetric: (p_1 *better* p_2) and (p_2 *better* p_1) $\Rightarrow p_1 = p_2$, since $t_1 \leq t_2$, $QoS_1 \geq QoS_2$, $t_1 \leq t_2$ and $QoS_1 \geq QoS_2$ imply that $t_1 = t_2$ and $QoS_1 = QoS_2$;
- *better* is transitive: (p_1 *better* p_2) and (p_2 *better* p_3) $\Rightarrow$ (p_1 *better* p_3), since $t_1 \leq t_2$, $QoS_1 \geq QoS_2$, $t_2 \leq t_3$ and $QoS_2 \geq QoS_3$ imply that $t_1 \leq t_3$ and $QoS_1 \geq QoS_3$. Consequently, *better* is an order relation. However, it is easy to find two proposals $\langle t_1, QoS_1 \rangle$ and $\langle t_2, QoS_2 \rangle$ that are not related, that is, ($t_1 > t_2$ and $QoS_1 \geq QoS_2$) or ($t_1 \leq t_2$ and $QoS_1 < QoS_2$), e.g. $\langle 2, 3 \rangle$ and $\langle 3, 4 \rangle$ in the example of Section 2.1. Thus, "*better*" is a partial order relation.

2.2.2. Proposal computation

In the following we present an algorithm that the QoS manager uses to produce the proposals to be provided to the user, given the available service projection of the system. We assume that the following operations are available on the lists of events and proposals. All lists are assumed to be ordered by increasing time.

Add(l, x): adds x to the end of the list l.
ExtractFirst(l, x): extracts the first element of the list l; the element extracted is x.
Merge($l, l1, l2$): merges the list l with the list $l1$. The result is $l2$ which is also ordered by increasing time.
GetLast(l, x): returns the last element, x, of the list l (no side effect).
Empty(l): Boolean; returns whether the list l is empty (no side effect).

For each tuple, $\langle t_i, QoS_i \rangle$, of the available service projection, the QoS manager checks whether QoS_i holds for a period of time, equal or longer than *length*, the period of the requested service. If the response is yes, then $\langle t_i, QoS_i \rangle$ may be considered as a potential proposal; otherwise the QoS manager will consider the other tuples with time values smaller than $t_i + length$ and bigger than t_i, in order to compute the maximum QoS which might hold for a period equal to *length* and starting from t_i. Each time a potential proposal is produced, it will be presented to the user only if there is no "*better*" proposal already presented.

Algorithm

```
Input:
  asp = [⟨$t_1$,$QoS_1$⟩,⟨$t_2$,$QoS_2$⟩, . . . , ⟨$t_n$,$QoS_n$⟩]
  (see Section 2.1);
  length indicates the length of the requested service;
Output: proposals
Variables:
  $m$, $m_1$ are integers;
  $x$, $y$ are proposals;
Initialization:
  $m := 1$;
  proposals: = [ ] (the empty list);
(1) while $m \le n$ do
  (1.1) if $t_m + length \le t_{m+1}$ then
      $x := \langle t_m, QoS_m \rangle$
    else
      $m_1 := m$;
      while $(t_{m_1} \le t_m + length)$ do
        $m_1 := m_1 + 1$;
      endwhile
      $x := \langle t_m, minimum(QoS_m, QoS_{m+1}, QoS_{m_1-1}) \rangle$;
    endif
  (1.2) GetLast(proposals, y);
  (1.3) if x is not comparable with y (in respect to better) then
      Add(proposals, x)
    endif
  (1.4) $m := (m+1)$;
  endwhile
end
```

In response to a service request issued by the user, the *proposal computation algorithm* will always return at least one proposal, $p = \langle t, QoS \rangle$, which meets the user requirements in terms of QoS; in the worst case, t corresponds to the first time (which may be very late) when all the resources of the system become available for a sufficiently long period of time. We assume, however, that (1) the user asks for a QoS that does not exceed the maximum system capacity, e.g. the user will not ask for throughput of 20 Mbits/sec over Ethernet which has a maximum speed of 10 Mbits/sec, and (2) there is no upper bound for the time for which proposals may be produced.

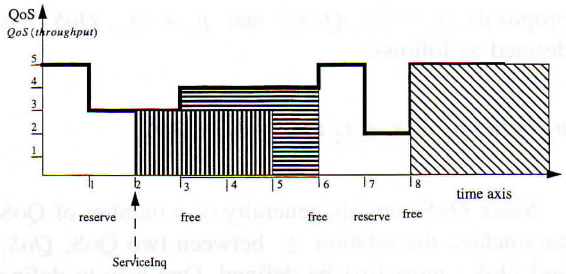


Fig. 4. Three proposals produced by the PC algorithm given the available service projection shown in Fig. 3.

Furthermore, no feasible proposal, for a given available service projection, can be "*better*" than any proposal returned by the algorithm. This means that the algorithm produces only the "best" proposals. A proof of this statement can be found in [17].

Example

Taking the available service projection shown in Fig. 3, and *length* = 3 as input, the *proposal computation algorithm* returns the following proposals (see Fig. 4): *proposals* = [⟨2, 3⟩, ⟨3, 4⟩, ⟨8, 5⟩]. Without the introduction of the relation *better*, the QoS manager would return two more (not-useful) proposals: ⟨6, 2⟩ and ⟨7, 2⟩.

2.3. Sharing of resources

In this section we show how NAFUR may be used to obtain optimal sharing of system resources.

Generally, a large number of users are served by a distributed system. One of the key problems induced is the sharing of the limited system resources. This becomes crucial when delivering the same information to more than one user, e.g. video-on-demand. It is argued in [16] that the use of multicast communication is the best way to serve more users in video-on-demand systems. The use of a multicast facility requires that the users want to receive the *same information*, e.g. the same movie, at the *same time*. However, the users are unlikely to ask for the same service at the same time.

NAFUR provides a suitable mechanism to encourage the sharing of system resources using the multicast facility. For this purpose we assume that the QoS manager keeps enough information about the users and their future resource reservations. Examples of this knowledge are: the location of users, e.g. users access lines, the service requested by a given user, e.g. to deliver the movie "Casablanca", the time the service will be provided, and the QoS requested. Upon receipt of a new service request, the QoS manager computes the QoS available at present and at certain future times. Some of these times may correspond to the starting times of the same service requested by other users for whom the resources are already reserved. When the QoS manager returns the proposals, if the user selects a proposal for which the required resources (or part) are already reserved (for another user), only a part of the required resources are reserved; no additional resources are reserved if the users of interest have the same access lines. To encourage users to select such proposals, the QoS manager may propose reduced cost.

To illustrate the operation of the QoS manager to support the sharing of resources, let us consider the example shown in Fig. 5. We assume the following scenario: (1) the resources are reserved (in the future) to play the movie "Casablanca" (with a desired QoS) for *user-1* at 20:30, and (2) *user-2* (respectively *user-5*) asks to play the movie "Casablanca" (with the same QoS) at 20:00 (respectively 20:15). The QoS manager produces the following results:

(a) One of the proposals returned to *user-2*, is to play the movie "Casablanca" at 20:30. If *user-2* selects this proposal, no additional resources will be reserved to support this proposal; the resources reserved for *user-1* can be shared with *user-2*, since they use the same access line.

(b) One of the proposals, returned to *user-5*, is to play the movie "Casablanca" at 20:30. If *user-5* selects this proposal, only a part of the required resources will be reserved to support this proposal; a certain amount of resources reserved for *user-1* (the resources of the video server and network-1) will be used to serve *user-5*, since the system use the multicasting facility to deliver "Casablanca" to *user-1* and *user-5*.

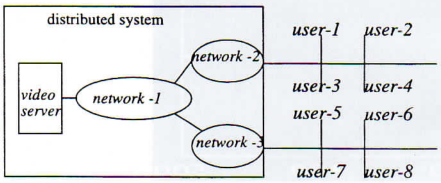


Fig. 5. An example of video-on-demand environment.

The definition of mechanisms and policies required to implement the mechanisms of resource sharing, e.g. multicast, are out of scope of the paper. However, such mechanisms may be used to compute the available service projection of the system. For example, the information (contained in the available service projection) which indicates that at time 3, the available throughput is 4 Mbits/s (see Fig. 3) does not necessarily mean that we have 4 Mbits which are not used at time 3. For example, this may mean that the 4 Mbits are reserved for a user, e.g. *user-1*, which has the same access line, who asked the same service with similar quality as the new request. Since we assume that the QoS manager knows about the available service projection, the use of these mechanisms does not affect the generality of NAFUR.

2.4. Performance evaluation

To evaluate the performance of NAFUR, we performed a number of simulations where we considered a video-on-demand system as a single module (as described in Section 2); the system is characterized by its maximum capacity R and the length of the movie is uniformly (randomly) distributed between 60 min and 90 min.

For the sake of simplicity and clarity, we assume that when the user requests to play a movie, the system (when using NAFUR) returns either an immediate acceptance or a single offer which represents a delayed movie presentation with the requested QoS (a future proposal); whereas, without NAFUR, it returns an acceptance or a rejection.

2.4.1. User behavior modelling

The user behavior is captured by the following parameters:

– *User request type:* indicates the class of QoS the user asks for. We assume that we have three classes *1*, *2*, and *3* which correspond to Q_1, Q_2, and Q_3 respectively. Q_1, Q_2, and Q_3 can be supported if 0.0050% of R, 0.0070% of R, and 0.0030% of R is available, respectively. The users will request the more popular class, *1*, more often; the probability that a user requests the class i is given by p_i. The following service request type pattern is assumed: $p_1 = 0.8$, $p_2 = 0.1$, and $p_3 = 0.1$.

– *User request pattern in time:* indicates the distribution of user requests over a day; this distribution presents a peak during the evening when most users ask to play a movie. A normal distribution, characterized by its mean ($\lambda = 3.5$) and its variance ($\sigma = 60$), is selected to model the evolution of this parameter.

– *Maximum delay parameter (MDP):* indicates the maximum difference (between the time the request is made and the starting time of a delayed presentation) which is acceptable by the user; a value of 0 for this parameter means that the user does not accept any delayed presentation of the requested movie.

– *Movie selection pattern*: indicates how users select one of the available movies (50 different movies). We assume that most popular movies are most often requested; the following is a default selection pattern: 80% of users selects the five most popular movies ($i = 1, ..., 5$); 15% of users selects the 20 less popular movies ($i = 6, ..., 25$); 5% selects the 25 least popular movies ($i = 26, ..., 50$).

2.4.2. Simulation results

The main metric we adopted for evaluation and comparison was the *rejection probability: rejection probability* = (number of rejections)/(number of service requests); a rejection corresponds to a rejection initiated by the system or a rejection initiated by the user who does not accept a delayed presentation (in this case *MDP* is smaller than the difference between the time of the request and the time of the future proposal returned by NAFUR).

Because of lack of space, we only describe the impact of two parameters on the performance of our system: the time the requests are made and the maximum delay parameter (MDP). A more detailed simulation study of our system can be found in [18]. For the experiments described below we assume that the number of users is 1000. For Fig. 6 through 8 we use the following notations: curves denoted by U correspond to user requests issued; curves denoted by S correspond to NAFUR without multicast; curves denoted by SM correspond to NAFUR with multicast; and curves denoted by V correspond to a system not using NAFUR.

In Figs. 6 and 7 the *X*-axis indicates the time in minutes (e.g. if 0 represents 17:00, then the peak of user requests is around 20:00); the *Y*-axis indicates the number of requests made and accepted in the last 20 minutes. Fig. 6 shows the number of user requests made and the number of requests accepted. The number of accepted user requests is much higher when using NAFUR; particularly, this holds during the peak of the request distribution (between 120 and

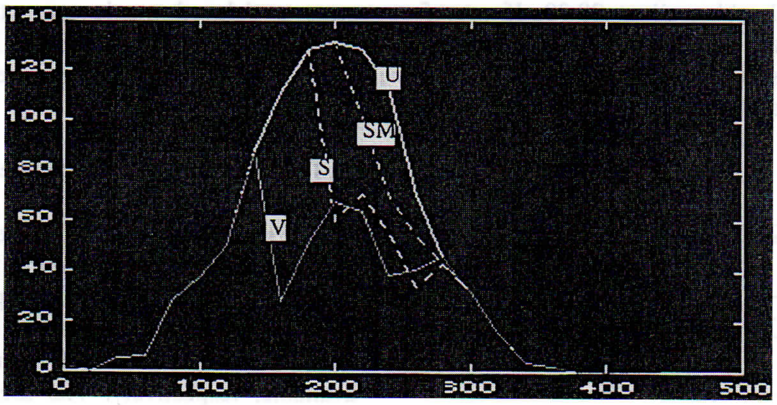


Fig. 6. User requests issued and user requests accepted.

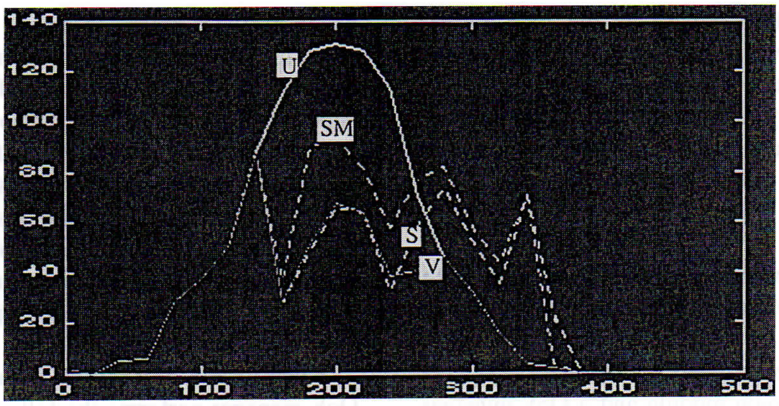


Fig. 7. User requests issued and users served.

280). For this experiment we assumed that the users do not accept presentations which are delayed more than 1 hour (60 minutes).

Fig. 7 shows the number of requests made and the number of requests served (started in the last 20 minutes). A large number of requests rejected when not using NAFUR are scheduled for future presentations when using NAFUR. The system using NAFUR is much better at handling a large number of requests made over a relatively short period of time.

It is obvious that NAFUR with multicast facility performs much better than NAFUR without such a facility; the use of multicast communication is the best way to serve more users in a video-on-demand system. This is due to the sharing of (server and network) resources between a number of users asking for the same movie, as explained in Section 2.3.

In Fig. 8, the *X*-axis indicates the maximum delay parameter; the *Y*-axis indicates rejection probability multiplied by 100. The figure shows that the blocking probability decreases when the value of the maximum delay parameter (MDP) increases; this is true when using NAFUR since a system not using NAFUR is not affected by varying MPD. The impact of MPD is most significant when using SVoD with multicast facility. One may argue that it is not realistic to assume that users will accept delayed presentations with a delay of 30 or 60 minutes; nevertheless, we believe that users will prefer to get a feedback from the system about the status of their request [18].

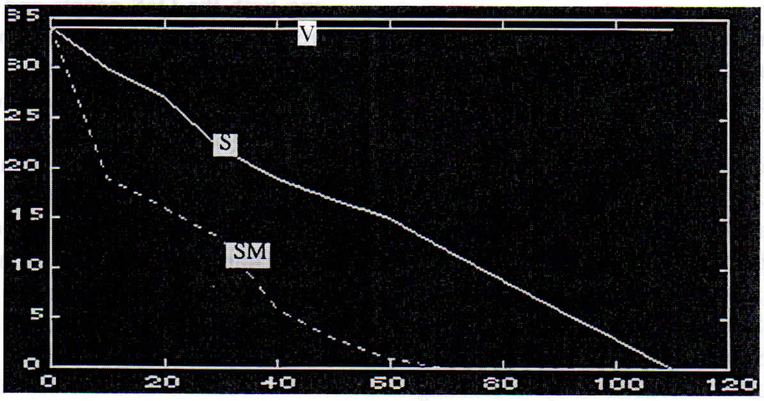


Fig. 8. Rejection probability versus maximum delay parameter.

NAFUR provides the feedback on the *time the requested presentation can start* and the *quality of the presentation*. We think that a good number of users will be attracted by money discounts for delayed presentations. Last and not least, users may book in advance in order to obtain a reservation.

3. A simple hierarchical negotiation of QoS with present and future service sessions

In Section 2, we assumed that a distributed system consists of a single module and a QoS manager (Fig. 2); this assumption is not realistic since a distributed system consists of a number of components, e.g. networks and servers. That is, a set of components are required to provide the QoS associated with a requested service. For example, in the context of a news-on-demand service, data is read from the disk, stored in buffers at the sender side, transmitted over the network and again stored in receiver buffers. Often, the signal will have to be decoded before it can be presented to the user. To provide QoS guarantees, each of these components should commit to provide a certain level of QoS.

The main emphasis in this section is on the definition of a framework which will support QoS management, especially QoS negotiation, in a distributed system. We introduce the components of the framework, namely the QoS managers and the QoS agents [15], and we define the interactions of these components to support the negotiation of QoS.

3.1. A framework for QoS management

Generally, there are three steps during the lifetime of a session, and particularly of a multimedia session: the establishment phase, the active phase and the clearing phase. During the three phases, QoS management is required to ensure that the requirements of the users are satisfied. Therefore, proper QoS must be negotiated during the establishment of the session, and controlled (ideally maintained) during the active phase. Our QoS management framework consists of QoS managers and a collection of QoS agents. A QoS manager is an entity which supports QoS management functions by interacting with the QoS agents, while a QoS agent is an entity which provides means to handle all the information about the performance and the functional behavior of a given component. For sake of simplicity, in this paper we consider only linear configurations of system components, which are sufficient to describe presentational MM applications, such as news-on-demand. However, the algorithms described in this paper can be easily adapted to be used for complex configurations (combination of linear configurations) which are required for other applications such as teleconferencing systems [21].

Upon receipt of a service request from the user, a QoS manager performs the following steps:

1. To identify the involved entities, $C_1, C_2, \dots, C_n$, in the provision of the service, e.g. database server, decoder, presentation device and network.
2. To determine the flow of data through these components. We will use the following notation to represent sequential data flow through the components: C_1–C_2–…–C_n, indicating that data flows from C_1 through C_2 through … and ends in C_n.
3. To determine the QoS parameters that can be provided by each component, by interacting with the corresponding QoS agents. We assume that the QoS agent of a given component maintains the available service projection and the resource reservation state of the components (as explained in Section 2).
4. To check that these levels of QoS allow to support the requested (end-to-end) QoS (see Eqs. (1)–(3) below).
5. To confirm the reservation of the corresponding resources in the different components by interacting with the QoS agents and to start the session.

More specifically, four main performance-oriented QoS parameters [10,1,19] have been identified for specifying distributed MM application requirements: delay, jitter, loss rate, and throughput. We note that the jitter is derived from the delay parameter; jitter is defined as the delay variation. Furthermore, it was reported that the most suitable location to deal with jitter guarantees is the sink component [20]. Consequently, the main performance parameters to consider are: throughput, delay, and loss rate.

To compute end-to-end QoS (*throughput, delay, lossrate*) relative to a service request based on the QoS characteristics (*throughput*$^{(i)}$, *delay*$^{(i)}$, *loss-*

rate$^{(i)}$) of the components C_i, we make use of the concatenation functions $G_t()$, $G_d()$, and $G_l()$:

$$throughput = G_t\left(throughput^{(1)}, \ldots, throughput^{(n)}\right) = \underset{i}{minimum}\left(throughput^{(i)}\right), \quad (1)$$

$$delay = G_d\left(delay^{(1)}, \ldots, delay^{(n)}\right) = \sum_{i=1}^{n} delay^{(i)}, \quad (2)$$

$$lossrate = G_l\left(lossrate^{(1)}, \ldots, lossrate^{(n)}\right) = 1 - \prod_{i=1}^{n}\left(1 - lossrate^{(i)}\right). \quad (3)$$

3.2. Hierarchical negotiation of service quality

The general hierarchical negotiation scenario (Fig. 9) is as follows: A user requests a new instance of the application, including specific QoS parameters, to the appropriate QoS manager. The QoS manager will first determine a configuration of system components that should be used for providing the end-to-end service for this particular instance of the application [15,21]. Each of these components may either be a simple component, as described in Section 3.1, or a composition of several simpler components. A simple component will be represented by its QoS agent, while a composition will be represented by another QoS manager.

In the second phase of the negotiation, the QoS manager will inquire (using ServiceInq function described in Section 2) the available service quality from each of the components participating in the configuration for this instance of application. Based on the obtained results, it will provide the user (or the QoS manager higher up in the hierarchy) with a

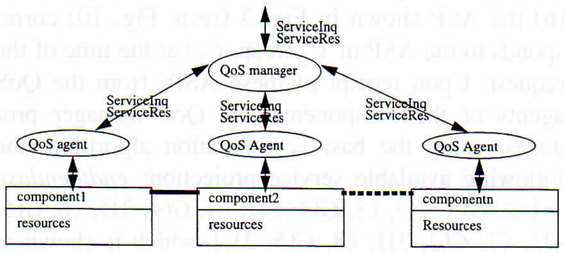


Fig. 9. Example of QoS manager–QoS agent interactions.

set of possible reservations that could be made for the requested service.

During the third phase, the user (or the QoS manager higher up in the hierarchy) makes a service reservation based on one of the suggested possibilities (using ServiceRes function described in Section 2). If the reservation starts immediately, the active phase of the session starts; otherwise the active session will start at the time for which the reservation is made.

We note that a QoS manager that invokes the ServiceInq operation on its components in order to determine what kind of service quality could be provided in response to a request made by a user, may possibly ask for the maximal quality of service which would be available. In the case of additive service qualities, such as delay, the manager may counterbalance a bad performance of one component by requesting a high QoS from another component that is able to provide a larger service quality.

3.3. Hierarchical computation of available service projection

The QoS manager receives the available service projections from the QoS agents possibly via some lower level QoS managers. Before producing the set of proposals to present to the user, the QoS manager should combine these available service projections to produce the available (end-to-end) service projection at the user interface. This information will then be used by the *Proposals Computation* (*PC*) *algorithm* (see Section 2.2.2) to produce proposals to be presented to the user.

3.3.1. Combination of available service projections

Given the available service projections (*asp*) produced by the QoS agents of interest, the QoS manager should be able to compute the available end-to-end service projection. This computation is performed by the *Basic Combination* (*BC*) *algorithm*. The basic idea is that the aggregated QoS is computed each time there is a change (transition) in the *asp* (available service projection) of one of the involved components; the concatenation functions $G_t()$, $G_d()$, or G_l (see Section 3.1) are used for this purpose.

Basic Combination (BC) algorithm

The initial value of the list *Masp* is the result of merging the available service projections of the involved components. The time (we call *tlast*) associated with the first tuple of *Masp* becomes the current time (initially, the current time is the starting time *starttime*); the algorithm extracts all the tuples of *Masp* for which the associated time is equal to the current time; then, it sets $Q^{(i)}$ (for $1 \le i \le n$) to the QoS provided by the component C_i at the current time ($Q^{(i)}$ represents a QoS variable); the aggregated QoS $G(Q^{(1)}, \ldots, Q^{(n)})$ is computed using the concatenation functions; $\langle tlast, G(Q^{(1)}, \ldots, Q^{(n)})\rangle$ becomes a tuple of the aggregated available service projection. This is repeated until *Masp* becomes empty.

Algorithm

Input:

$asp^{(1)}, \ldots, asp^{(n)}$ the available service projections of $C_1, \ldots, C_n$. We associate to each element of $asp^{(i)}$, for $1 \le i \le n$, an attribute i which identifies the corresponding component. For example, an element of $asp^{(i)}$ will be written as $\langle t, QoS\rangle^{(i)}$;

starttime is the desired starting time for the service;

Output:

endtoendasp which is a list of tuples $\langle time, QoS\rangle$, where *Qos* corresponds to the (end-to-end) aggregated QoS that can be provided at *time*.

Variables:

$Q^{(i)}$, for $(1 \le i \le n)$, may assume any QoS value;

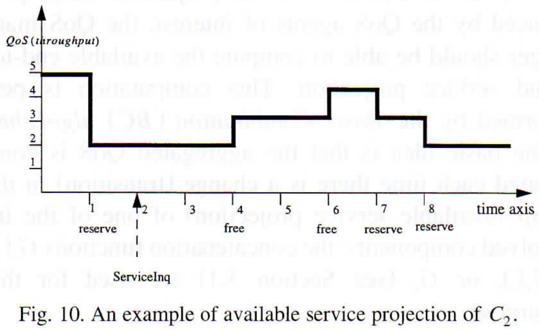


Fig. 10. An example of available service projection of C_2.

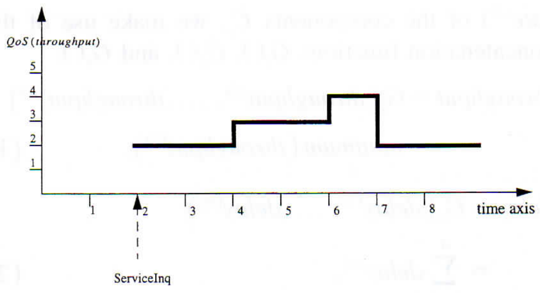


Fig. 11. The available service projection (ASP) produced by the BC algorithm given the ASPs shown in Fig. 3 and Fig. 10.

Masp is a variable representing available service projections;

tlast is a time variable;

endtoendasp is a list representing the (end-to-end) available service projection;

```
Initialization:
    $tlast := starttime$;
    $endtoendasp := [\,]$;
    Merge($asp^{(1)}, \ldots, asp^{(n)}, Masp$);
(1) while not (empty($Masp$)) do
    (1.1) ExtractFirst($Masp, \langle t, QoS\rangle^{(i)}$);
    (1.2) if $t \ne tlast$ then
        Add($endtoendasp, \langle tlast, G(Q^{(1)}, \ldots, Q^{(n)})\rangle$);
        /* when $t \ne tlast$, the initialization of the
        $Q^{(i)}$, for $(1 \le i \le n)$, was already performed
        in previous executions of Step (1.3) */
        $tlast := t$;
    endif
    (1.3) $Q^{(i)} := Qos$;
    endwhile
    end
```

Example, Let us assume that (a) two networks with a maximum bandwidth of 5 Mbits/sec, called C_1 and C_2, are required to support the user request, and (b) the ASP shown in Fig. 3 (resp. Fig. 10) corresponds to the ASP of C_1 (resp. C_2) at the time of the request. Upon receipt of these ASPs from the QoS agents of the components, the QoS manager produces, using the basic combination algorithm, the following available service projection: *endtoendasp* $= [\langle 2, G(3, 2)\rangle, \langle 3, G(4, 2)\rangle, \langle 4, G(4, 3)\rangle, \langle 6, G(5, 4)\rangle, \langle 7, G(2, 3)\rangle, \langle 8, G(5, 2)\rangle]$, which is shown in Fig. 11.

3.4. An optimized hierarchical negotiation of QoS with present and future service sessions

The interactions between the QoS agents and the QoS managers, presented in Section 3.3, are not optimal in terms of the information exchanged and the response time obtained (to produce the proposals to be presented to the user). Indeed, the QoS agents send the complete information of the available service projections to the QoS manager without any pre-processing; this information could be reduced if pre-processed by the QoS agents. That is, a QoS agent may process its available service projection in a way to produce a reduced amount of information (to be sent to the QoS manager) which may be easily used by the QoS manager to produce the proposals to be presented to the user.

Upon receipt of the available service projection of the involved components the QoS manager uses the *BC algorithm* to produce the (end-to-end) available service projection. This allows the QoS manager to know about the end-to-end QoS that can be provided at a given time; however, it cannot, directly, know whether this QoS is available for the duration of the requested service; for this reason it uses the *proposals computation algorithm* to build the proposals to be presented to the user. However, if QoS agents send an information about the duration of the (local) availability of a QoS starting from a given time, the QoS manager can directly produce the combined QoS for the user. This means that a QoS agent should process its available service projection to produce and send ''internal proposals'', p, defined as follows: $p:=\langle t, QoS, tmax\rangle$ where $tmax$ represents the maximum period over which the component can support QoS starting from t.

For sake of clarity let us consider the following example. We assume that a user issues a service request with a desired QoS to start at a certain time t_1 for a period of *length*. Upon receipt of the user request, the QoS manager sends service requests to two QoS agents of interest (the service requested might require these two components: C_1 and C_2). In response, two available service projections (ASP), $asp^{(1)}$ and $asp^{(2)}$, are received by the QoS manager. Using the approach described in Section 3.3, the QoS manager starts by considering the first tuples $p^*=\langle start\text{-}time, QoS^*\rangle \in asp^{(1)}$ and $p^\dagger=\langle start\text{-}time, QoS^\dagger\rangle \in asp^{(2)}$ to produce the first tuple, $\langle starttime, G\ (QoS^*, QoS^\dagger)\rangle$, of the (end-to-end) available service projection. If the QoS manager has the information that QoS^* and $QoS^\dagger$ might be supported over $[starttime, starttime+length]$, then it will produce directly the proposal $\langle starttime, G\ (QoS^*, Qos^\dagger)\rangle$ to be presented to the user; without this information, the QoS manager should use the *proposals computation algorithm*. This scenario is applicable not only for *starttime* but also for any later time value. Consequently, if the QoS agents provide the QoS manager with this (added-value) information concerning the length of time over which a given QoS can be supported, then the complexity of the operation of the QoS manager will be decreased and the quantity of information to be sent by the QoS agents will be reduced (since the tuples $\langle t, QoS\rangle$ of the available service projections for which the QoS does not hold for a period of at least *length* will not be sent to the QoS manager).

In [17] we presented two algorithms, namely the so-called *Internal Proposal Computation (IPC) algorithm* and the *Proposal ComBination (PCB) algorithm* which realize these ideas. The *IPC algorithm* is used by the QoS agents to produce ''internal proposals'' ($[\langle t, QoS, tmax\rangle, \ldots]$) from the available service projections. Upon receipt of the ''internal proposals'' built by the QoS agents, the QoS manager uses the *PCB algorithm* to produce proposals to be presented to the user. It is worth noting that the *PCB algorithm* and the *IPC algorithm*, together, produce the same proposals as the *BC algorithm* (see Section 3.2) together with the *proposals computation algorithm* (see Section 2.2).

3.5. Constraining the number of proposals produced by a QoS agent

In response to the QoS manager, a QoS agent produces a set of proposals; the number of proposals depends only on its available service projection. Even with the optimized approach described in Section 3.4, a QoS agent may produce a huge number of internal proposals which may not all be ''useful'' to produce the proposals to be presented to the user. One may think that, if we bound the number of proposals to be produced by the QoS agents, the operation of NAFUR will be further optimized. Indeed, the information sent by the QoS agents will be reduced, and the complexity of the operation of the QoS manager will also be reduced since it will process a smaller quantity of information. However, with a bounded number of proposals there is no guarantee of producing all relevant proposals; in particular, in certain situations, no proposal may be produced even though some proposal would be possible.

For example, let us assume that, after executing the algorithms described in Section 3.4, the components C_1 and C_2 return $proposals^{(1)} = [\langle(t_1^{(1)} = 13{:}00), QoS_1^{(1)}, (tmax_1^{(1)} = 25 \text{ min})\rangle, \langle(t_2^{(1)} = 13{:}20), QoS_2^{(1)}, (tmax_2^{(1)} = 30 \text{ min})\rangle, \langle(t_3^{(1)} = 14{:}00), QoS_3^{(1)}, tmax_3^{(1)}\rangle]$ and $proposals^{(2)} = [\langle(t_1^{(2)} = 14{:}00), QoS_1^{(2)}, tmax_1^{(2)}\rangle, \langle(t_2^{(2)} = 14{:}15), QoS_2^{(2)}, tmax_2^{(2)}\rangle]$, respectively; then, a possible proposal to be presented to the user is $\langle 14{:}00, G(QoS_3^{(1)}, QoS_1^{(2)})\rangle$. However, if we limit the number of proposals to be returned by the QoS agents to two (the first two tuples in each list), then we note that at 14:00 the QoS manager does not know the QoS that can be provided by C_1 since $t_1^{(1)} + tmax_1^{(1)} = 13{:}25 < t_1^{(2)} = 14{:}00$ and $t_2^{(1)} + tmax_2^{(1)} = 13{:}50 < t_1^{(2)} = 14{:}00$; consequently, the QoS manager produces no proposal to be returned to the user.

If the limited number of proposals to be sent by the QoS agents are carefully selected then the probability that the QoS manager produces one or more proposals will increase. In [17] we presented a *proposal selection (PS) procedure* which allows the QoS agents to select "suitable" proposals to be sent to the QoS manager. The main idea of the *PS procedure* is that QoS agents communicate some information about their QoS availabilities to the QoS manager, either periodically or when significant changes occur to their current load. When the QoS manager receives a user request, it uses this information to specify the bounds of the starting time (and other parameters) of the proposals to be sent by the QoS agents. Obviously, only a limited number of proposals will satisfy such requirements.

3.6. Support of multiple simultaneous service requests

As described in Section 3, the proposed algorithms support the processing of only one service request at a given time. That is, we did not describe the behavior of a QoS agent when it receives a new ServiceInq while it is waiting to receive a reservation (ServiceRes) for a previous ServiceInq. The problem with the support of multiple simultaneous service requests is that each QoS agent reserves a certain amount of its resources to support the proposals produced in response to a ServiceInq. Then, when the QoS agent receives a new ServiceInq, it has not enough information to process it, because it does not know the proposal which may be selected by the user in respect to the previous ServiceInq.

Three approaches can be considered to deal with this issue: (1) the pessimistic approach: to use a locking protocol to keep the resources reserved until the corresponding ServiceRes is received by the QoS agent (the upcoming ServiceInq are ignored or queued); (2) the optimistic approach: to process the upcoming ServiceInq hoping that all the service requests can be supported; or (3) a hybrid approach: to make some assumptions on the previous ServiceInq in order to process the upcoming ServiceInq hoping that these assumptions become true. A detailed description of these three approaches and a discussion how they can be used by NAFUR is given in [17]

4. The case of unknown service durations

For a range of multimedia applications, it is difficult to predict the duration of a session. For example, collaborative systems assume that discussions are somewhat unstructured and therefore it is almost impossible to know, in advance, the (accurate) duration of a collaborative session. In such a situation, the reservation algorithms described above cannot be used directly, since they assume that the service duration is known when the service request is made. We discuss in the following how these algorithms can still be used, assuming that the effective service durations can be approximately estimated when the request is made. We assume that for each requested service we know its estimated duration. Its value, *estimatedlength*, may be (a) set by the user, or (b) computed based on statistics gathered while running the applications (note on known length for video-on-demand). We use *estimatedlength* instead of *length* in the algorithms described earlier. Three cases are identified: (1) the effective duration, noted by *effectivelength*, is equal to the estimated duration (exact termination); (2) the effective duration of the application is longer than the estimated duration (delayed termination); and (3) the effective duration of the application is shorter than the estimated duration (premature termination). Premature termination and delayed termination raise some issues related to QoS guarantees and resource utilization, discussed below, while exact termination corresponds to the situation discussed in Sections 2 and 3.

In the case of delayed termination, there are no guarantees to support the requested QoS after the estimated period, i.e. over [*starttime* + *estimatedlength*, *starttime* + *effectivelength*], in the case of premature termination, resource utilization will be not optimal since the resources (reserved for the requested service) are not used up to the end of the reservation period, i.e. they remain idle during the period [*starttime* + *effectivelength*, *starttime* + *estimatedlength*]. We discuss in the following how one could deal with these issues.

4.1. Delayed termination

At the end of the reserved period, i.e. at the time *starttime* + *estimatedlength*, the QoS manager will perform a new negotiation to determine the QoS which might be supported for the service in question over the time period between the estimated termination and the effective termination. If there are enough resources to support the requested QoS over this time period, then the QoS agents have only to update their available service to reflect the new situation; otherwise, the user is notified that his/her requested QoS cannot be supported any more. Then, the user has the choice to abandon or initiate a renegotiation.

If we want to maintain QoS guarantees for services with delayed termination, a priority-based approach may be used. That is, to each service request is associated a priority parameter; the user specifies the desired priority for his/her request during the negotiation phase. Thus, if a high priority is associated with a service with delayed termination, the QoS manager may abort some low priority services which are currently provided (or may delay the starting time for low priority services which are scheduled to start in the future) if not enough resources are available to maintain QoS guarantees (for the high priority service) until the effective termination; this means that a higher priority service request is satisfied before low priority service requests (e.g. the priority parameter assumes two values: high and low). The time duration over which the resources are reserved at the time *starttime* + *estimatedlength* for the request with delayed termination is determined either by the system (based on statistical information), or by the user by means of renegotiation.

The cost is a key concept in such an approach. Without cost constraints, the users will always ask for high priority for their requests; the cost will limit the greediness of the users. The cost for high priority service requests should be significantly higher than the cost for low service requests. More specifically, the cost to be paid by a user is defined as a function of the amount of resources used, the time period the resources are held, and the priority of the service request. For the users who under-estimated *estimatedlength* the situation depends on the priority:

(a) When a high priority service request is accepted, the QoS manager should provide QoS guarantees even after the originally estimated period. A simple way to meet such stringent requirements is to abort some low priority services; however, more sophisticated solutions may be used. As an example, for a given high priority service request, the QoS manager could reserve the resources over [*starttime*, *starttime* + *estimatedlength* + *delinquenttime*], where *delinquentime* is a time value which is maintained by the QoS manager. The computation of delinquentime depends on several factors such as, the service nature and the general behavior of the users, e.g. a user never uses the requested service more than 10 minutes, and is based on past experience. If the *delinquentime* used by the QoS manager is too small (*estimatedlength* + *delinquenttime* < *effectivelength*) the abortion of low priority services may be required.

(b) For low priority service requests, by *starttime* + *estimatedlength* − *reactiontime* the QoS manager sends a warning to the user indicating that the requested service will still be provided for a duration equal to *reactiontime*, where *reactiontime* is a small time value determined by the QoS manager. *reac-*

tiontime can be computed as the average of the response times to perform a negotiation. Upon receipt of this warning, the user may ignore it, e.g. if he/she knows that he/she will end the session soon, or initiate a new service request. If the new request cannot be supported, the QoS manager asks the user whether he/she wants to increase the priority of his/her service request. If the response is yes, the QoS manager will process the request as described under (1) above; in this case, the user has to pay an extra-charge for (dynamic) priority increase.

4.2. Premature termination

To avoid the under-utilization of resources due to premature termination of certain service requests, we have only, for a component C, to update immediately its available service projection each time a service, using C's resources, ends (instead of waiting until $starttime + estimatedlength$). Thus, when new service requests are received by the QoS agents, the updated available service projections are used to produce more realistic proposals to be returned to the QoS manager.

This behavior of the QoS agents is simple; however, more sophisticated actions may be considered. For instance, the QoS agents may send notifications informing the QoS manager about the new situation (the new ''internal proposals'' or the new available service projection). The QoS manager applies the algorithms, as described above, with the updated ''internal proposals'' to reschedule some services: upon the receipt of the QoS agent(s) notification(s), the QoS manager initiates a new negotiation for some scheduled, e.g. active or ahead scheduled, services. Then, the user is informed about the new situation (by means of proposals); the user has the choice to accept or ignore the new proposal(s). If the QoS manager does not receive a response from the user within a certain period of time (*reactiontime*), then it will assume that the new proposal(s) is rejected.

In order to discourage users from making unnecessarily long service requests, the following charging policy may be adopted. The users who over-estimated *estimatedlength* will be charged at the regular rate for the effective usage of resources (over the effective length of the service) and at a ''reservation rate'' (which is lower than the regular rate) for the reserved resources not used (over the time period between the effective termination and the estimated termination of the service). We believe that this policy is essential if we want to improve the availability of the system.

5. Conclusion

This paper describes work on a new QoS negotiation approach with future reservations (NAFUR) that decouples the starting time of the requested service from the time the service request is made. It is assumed that the duration of the requested service requested is known. NAFUR allows to compute the QoS that can be supported at the time the service request is made, and at certain later times carefully chosen. For example, if the requested QoS cannot be supported at the time the service request is made, the proposed approach allows to compute the earliest time, when the user can start the service with the desired QoS. We also defined some rules and policies which could be used with the proposed algorithms in situations where the service duration is not know when the request is made.

To the best of our knowledge, negotiation with future reservations is not available within any of the existing schemes and protocols for QoS negotiation. However, there is some similarity with the approach of Nussbaumer et al. [22] in the context of Community Access TV (CATV) for on-demand MM distribution, which allows the users to wait a bounded time period if the resources are not available at the time of the request. But, this approach has the following characteristics: (1) it does not consider the concept of QoS; (2) it does not provide the user with the expected starting time (the user has to wait); (3) it does not provide any algorithms to produce proposals in response to the user's request; and (4) it considers a particular system (CATV).

We believe that NAFUR will help to increase (a) the flexibility of the system by providing the user with more choices; and (b) the system resource utilization and the availability of the system, by encouraging the sharing of resources, especially through multicasting. Furthermore, it can be easily used to support service requests made in advance, which is especially useful for scheduled multi-party communications, such as tele-conferencing and tele-teaching systems.

Last but not least, NAFUR provides the flexibility to incorporate (a) a range of resource reservation schemes and scheduling policies, and (b) a range of new system component technologies.

We are planning to implement NAFUR in the context of the CITR (Canadian Institute for Telecommunication Research) news-on-demand prototype [23] and to quantitatively evaluate the gain obtained by using NAFUR in the context of the news-on-demand application.

Acknowledgements

This work was 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] D. Anderson, R. Herrtwich, C. Schaefer, SRP: a resource reservation protocol for guaranteed-performance communications in the Internet, The International Computer Science Institute, Berkeley, 1991.

[2] C. Chou, K. Shin, Statistical real-time video channels over a multi-access network, Proc. High SPIE, 1994.

[3] A. Danthine, OSI95: high performance protocol with multimedia support on HSLANs and B-ISDN, 3rd Joint European Networking Conf., Innsbruck, 1992.

[4] D. Ferrari, D. Verma, A scheme for real-time channel establishment in wide-area networks, IEEE JSAC 8 (3) (1990).

[5] D. Kandlur, K. Shin, D. Ferrari, Real-time communication in multi-hop networks, Proc. 11th Int. Conf. on Distributed Computing Systems, 1991.

[6] B. Metzler, I. Miloucheva, K. Rebensburg, Multimedia communication platform: specification of the broadband transport protocol XTPX, CIO, RACE Project 2060, 60/TUB/CIO/DS/A/002/b2, 1992.

[7] C. Topolcic, Experimental Internet stream protocol: version 2 (ST-II), Internet RFC 1190, 1990.

[8] D. Hehmann, R. Herrtwich, W. Schulz, T. Schett, R. Steinmetz, Implementing HeiTS: architecture and implementation strategy of the Heidelberg high speed transport system, 2nd Int. Workshop on Network and Operating System Support for Digital Audio and Video, IBM ENC, Heidelberg, Germany, 1991.

[9] J. Ramaekers, G. Ventre, Quality of service negotiation in a real-time communication network, Technical Report TR-92-023, ICSI, Berkeley, CA, 1992.

[10] D. Ferrari, A. Banerjea, H. Zhang, Network support for multimedia, Technical Report 92-072, International Computer Science Institute, Berkeley, CA, November 1992.

[11] G. Kalkbrenner, T. Pirkmayer, V. Dornik, P. Hofmann, Quality of service in distributed hypermedia-systems, 2nd Int. Workshop on Principles of Document Processing, Darmstadt, April 1994.

[12] B. Kerherve, A. Vogel, G. Bochmann, R. Dssouli, J. Gecsei, A. Hafid, On distributed multimedia presentational applications: functional and computational architecture and QoS negotiation, Proc. High Speed Networks Conf., 1994, pp. 1–19.

[13] K. Nahrstedt, An architecture for end-to-end quality of service provision and its experimental validation, Ph.D. Thesis, University of Pennsylvania, 1995.

[14] W. Tawbi, E. Horlait, Expression and management of QoS in multimedia communication systems, Ann. Telecommun. (June 1994).

[15] A. Hafid, G.V. Bochmann, Quality of service adaptation in distributed multimedia applications, ACM/Springer Multimedia Systems J. (1997) (to appear).

[16] K. Almeroth, M. Ammar, Providing a scalable interactive video-on-demand service using multicast communication, Proc. ICCCN 94, San Francisco, CA, 1994.

[17] A. Hafid, QoS management in distributed multimedia applications, Ph.D. Thesis, University of Montreal, Montreal, Canada, 1996, pp. 96–133.

[18] A. Hafid, A scalable video-on-demand system using future reservation of resources and multicast communications: design and implementation, Comput. Comm. (1997) (to appear); a version is in Proc. Fifth IFIP Int. Workshop on QoS (IWQoS), New York, 1997.

[19] I. Miloucheva, QoS management for high speed transport architecture, Workshop on Distributed Multimedia Applications and QoS Verification, Montreal, Canada, June 1994.

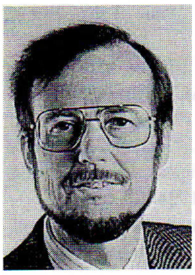
[20] C.J. Sreenan, Synchronization services for digital continuous media, Ph.D. Dissertation, University of Cambridge, UK, 1992.

[21] G.V. Bochmann, A. Hafid, Some principles for quality of service management, IEE Distributed Syst. Eng. J. 4 (1) (1997) 16–27.

[22] J. Nussbaumer, F. Schaffa, Capacity analysis of CATV for on-demand multimedia distribution, Proc. IASTED/ISMM Int. Conf. on Distributed Multimedia Systems and Applications, Honolulu, Hawaii, 1994.

[23] J. Wong, K. Lyons, R. Velthuys, G. Bochmann, E. Dubois, N. Georganas, G. Neufeld, T. Ozsu, J. Brinskelle, D. Evans, A. Hafid, N. Hutchinson, P. Inglinski, B. Kerherve, L. Lamont, D. Makaroff, D. Szafron, Enabling technology for distributed multimedia applications, IBM Syst. J. 36 (4) (1997) 489–507.

Abdelhakim Hafid is Assistant Professor at the Electrical & Computer Enggineering/Compter Science Departments (a joint appointment), University of Western Ontario, and a Research Director of the Advanced Communication Engineering Centre (venture established by UWO, Bay Networks, Bell Canada); he is also an Adjunct Professor at the University of Montreal, Department of Computer Science. He received his Masters and Ph.D. degrees in computer science from the University of Montreal on quality of service management for distributed multimedia applications in 1993 and 1996, respectively. From 1996 to 1997 he was a Researcher Staff Member at the Computer Research Institute of Montreal (CRIM), Telecommunications and Distributed Systems Division, working in the area of distributed multimedia applications. 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 Internet and multimedia networking.



Gregor von Bochmann is a professor at the University of Montreal since 1972 and holds the Hewlett-Packard-NSERC-CITI chair of industrial research on communication protocols. He is also one of the scientific directors of the Centre de Recherche Informatique de Montréal (CRIM), and a Fellow of the IEEE and ACM. Professor von Bochmann has worked in the areas of programming languages, compiler design, communication protocols, and software engineering and has published many papers and some books in these areas. He has also been actively involved in the standardization of formal description techniques for OSI communication protocols and services. From 1977 to 1978 he was a visiting professor at the Ecole Polytechnique Fédérale, Lausanne, Switzerland. From 1979 to 1980 he was a visiting professor in the Computer Systems Laboratory, Stanford University, California. From 1986 to 1987 he was a visiting researcher at Siemens, Munich, Germany. His present work is aimed at methodologies for the design, implementation and testing of communication protocols and distributed systems. Ongoing projects include applications to high-speed protocols, distributed systems management and quality of service negotiation for distributed multimedia applications.

Rachida Dssouli is professor in the Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal. She received the Doctorat d'Université degree in computer science from the Université Paul-Sabatier of Toulouse, France, in 1981, and a Ph.D. degree in computer science in 1987, from the University of Montreal. Professor Dssouli has been professor at the Université Mohamed 1er, Oujda, Morocco, from 1981 to 1989, and assistant professor at the Université de Sherbrooke, from 1989 to 1991. She is currently on Sabbatical at NORTEL, Ile des Soeurs. Her research area is protocol engineering and requirements engineering. Ongoing projects include incremental specification and analysis of reactive systems based on scenario language, multimedia applications and tests of timed communicating systems.