QoS Negotiation and Resource Reservation for Distributed Multimedia Applications

Kurt Rothermel, Gabriel Dermler, Walter Fiederer

CR-Klassifikation: C.2.4, D.4.7, H.5.1

**QoS Negotiation and
Resource Reservation for
Distributed Multimedia Applications**

K. Rothermel, G. Dermler, W. Fiederer

Fakultätsbericht 1996/09
Technical Report
Juli 1996

Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20 - 22
D-70565 Stuttgart

Abstract

Distributed multimedia applications require negotiation of quality of service (QoS) and resource reservation for distributed application components and communication links. Negotiation of QoS for an application session is a balancing process between the QoS specified by a client, the resource capabilities of the distributed system and the functional capabilities of the distributed application. Negotiation requires application level QoS descriptions and an end-to-end view spanning the whole distributed application. XNRP, introduced in this paper, is a protocol meeting these requirements. XNRP performs negotiation based on client specified QoS value ranges and is independent of application level QoS semantics. It allows QoS negotiation and resource reservation in three phases and supports arbitrarily interconnected flowgraphs of application components.

1 Introduction

Advances in the computer and communication technology have stimulated the integration of digital audio and video with computing, leading to the development of distributed multimedia systems. This class of systems combines the advantages of distributed computing with the capability of processing discrete media and continuous media, such as audio or video, in an integrated fashion. The capability of integrated multimedia processing not only enhances conventional application environments, but also opens the door for new and innovative applications ranging from on-demand teleservices and teleconferencing systems to distributed game and virtual reality applications.

Resource reservation is required in order to provide deterministic service for networked multimedia systems. In the past, various reservation protocols have been developed for the transport and network level (e.g, Tenet Protocol Suite [1], ST-II[2], RSVP[3], XTPX[4]). Resource reservation at this level aims at providing QoS guarantees for the real-time data transfer between end systems. QoS is specified in terms of parameters, such as packet size, packet rate and delay, jitter and loss rate. Those QoS parameters are generic in the sense that they do not specify media-specific stream characteristics, such as frame size, frame rate and color depth in the case of a video stream.

However, if we consider multimedia systems on a higher level - the application level - media specific-parameters have to be taken into account. Firstly, the QoS specified at the application level will include media-specific parameters, e.g., an observer of a video might want to specify the desired frame size. Secondly, multimedia processing elements, such as video sources, filters, and mixers, might have limited capabilities with regard to the stream “formats” they are able to consume and/or produce. For example, a filter might be able to consume a stream of type “MPEG_encoded -video” only up to a certain frame size. We call those limitations on media parameters format constraints. Thirdly, there might be various dependencies between streams concerning their media-specific parameters. For example, the frame size expected at a video mixer may vary, however the same frame size is required at each of its input streams. We call those dependencies stream relations. Finally, if resource reservation is done at the application level, in many cases media-specific parameters have to be taken into account to determine the resource requirements. For example, the computational resources required by a class of video filters directly depend on the frame size and frame rate rather than “bandwidth”.

When setting up sessions at the application level, stream-specific QoS requests, format constraints, stream relations as well as the resources available at set-up time are to be considered. Taking into account all this information the set-up process negotiates media-specific parameters and reserves the required resources. Moreover, set-up must be able to operate on rather complex structures since multimedia processing elements can be configured into arbitrarily structured flow graphs, which may be distributed over any number of end systems.

Only few application-level set-up protocols have been proposed so far. In [5] a negotiation protocol is proposed, which is limited to client/server settings (i.e. one/ or many recipients connected to a media source). In [6], the Negotiation and Resource Reservation (NRP) protocol was introduced. While this protocol already supports rather complex application topologies, there are still some limitations on the type of flow graphs supported: each flow graph has to be subdivided into a mixing and multicasting zone. NRP has been extended to XNRP, which now supports arbitrarily structured flow graphs.

XNRP is an application-level session set-up protocol for the deterministic service model. It is based on a deterministic transport-level service, which can be provided by one of the reservation protocols mentioned above. Though designed for a deterministic service, the proposed mechanisms for the negotiation media-specific parameters in complex flow graphs can be also applied in best-effort environments.

The remainder of the paper is structured as follows. In Section 2, we introduce the underlying application model, and then briefly describe our QoS architecture in Section 3. The concepts of stream relations, format constraints and variable filters are motivated and defined in Section 4 before XNRP is presented in section 5. First, an overview of the protocol is given, which is followed by a comprehensive protocol description and a detailed example. The paper concludes with a short summary.

2 Application Model

We briefly introduce our abstractions for modeling distributed multimedia applications. Similar concepts have been proposed by various other groups (e.g. [7], [8], [9]), including the consortium defining IMA MMS [10]. Here, we describe the terminology used for our CINEMA development platform [11].

In CINEMA, distributed multimedia applications are built by configuring so-called flow graphs, consisting of a number of component objects interconnected via links. Components are processing elements encapsulating functions for capturing, storing, presenting and manipulating continuous data streams. They are associated with ports, which allow them to communicate stream data. Flow graphs are configured by linking the ports of components, where a link is an abstraction of a unidirectional communication channel, conveying stream data from one output port to one or more input ports. We distinguish source components, associated with output ports only, sink components, which only have input ports, and intermediate components, which receive data from a number of input ports, perform some operation on the received data, and send the result via a number of output ports. Examples of the latter component type are filters, having one input port, and mixers, having several input ports.

Figure 1 shows an example video conferencing scenario. The flow graph is composed of two video source components connected to a picture-in-picture mixing component, which provides a data stream multicasted to two sink components. Prior to each sink component a filter component is included.

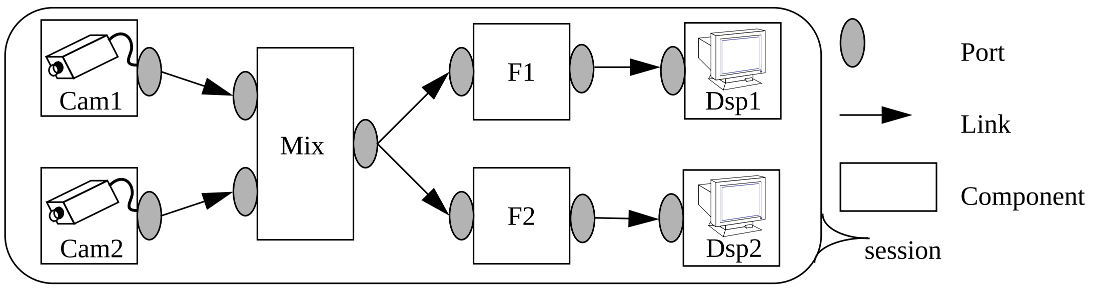

Figure 1 : Video Conferencing Scenario with Mixer and Multicast

In CINEMA, data streams are typed. For example, the media type of a stream may be "uncompressed_video", JPEG_encoded_video" or "G.711_encoded_audio". Each media type defines a set of so-called media parameters, which specify the characteristics of a particular stream instance. For example, "uncompressed_video" may be associated with parameters, such as frame rate, picture size and color depth, given that these parameters describe the degrees of freedom for a specific stream instance (see [10]). As streams are communicated via ports, port objects are typed and can only be linked if they are of compatible type. Type checking is performed during instantiation of application topologies.

The CINEMA system provides a service interface for configuring and controlling distributed multimedia applications. These services are employed by software entities, referred to as clients. Three services classes are provided, configuration management, session management, and stream control & synchronisation management. Configuration management allows a client to construct a distributed component topology [12], and session management provides for setting up (and tearing down) sessions, taking into account the client's QoS requirements. Sessions, which may encompass arbitrarily complex flow graphs, are the units of resource allocation for stream communication and processing. Finally, the control & synchronisation services enable clients to control and synchronise the flow of individual streams as well as client-defined groups of streams ([13], [14]).

3 QoS Architecture

The design of session set-up in CINEMA is based on a QoS architecture distinguishing between the client level, session level and transport level (Figure 2). The objects considered at the upper two layers are clients and flow graphs as described above. Link objects, which at the session level abstract from a concrete communication mechanism, employ some transport level mechanism whenever components residing on different sites are to be linked. According to the layered architecture, three levels of QoS are considered: client-level QoS (C-QoS), session level QoS (S-QoS) and transport level QoS (T-QoS).

At the highest level, C-QoS is specified in a manner appropriate to end users, and hence is end user-specific. For example, it is conceivable that for end users terms such as "high, medium, low" quality are sufficient to express QoS requirements for a video presentation. Of course, the semantics of those terms highly depend on the particular application class, and hence may vary from application to application. It is even conceivable that for a given application different types of C-QoS specs are provided for different classes of users.

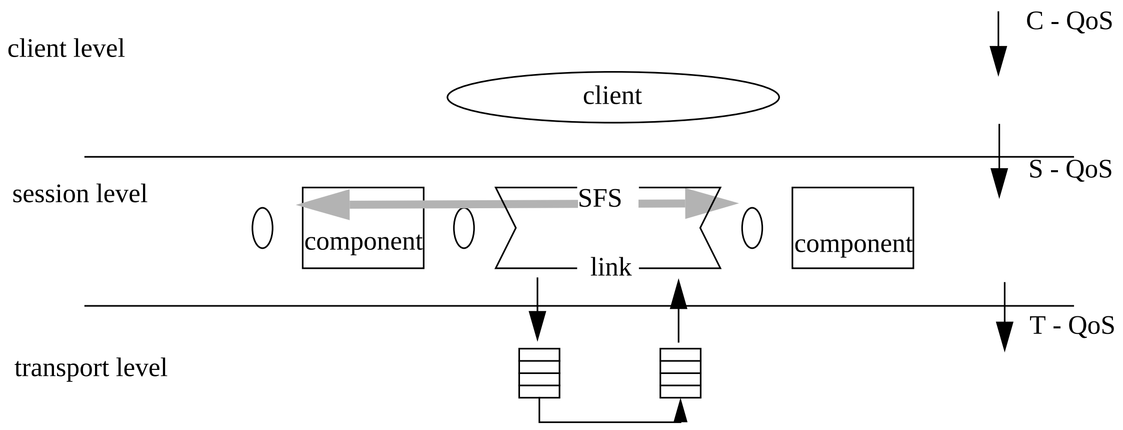

Figure 2 : QoS Architecture

The QoS specified at the session level interface is media-specific. S-QoS of a stream is defined by the media parameters of the stream's type and a number of generic parameters, such as delay, jitter and loss rate. The media-specific parameters of S-QoS allow to express the desired media-specific characteristics of the data stream, such as frame rate, picture size, color depth for an "uncompressed_video" stream. S-QoS is specified at the (typed) input ports of sink components. A session may comprise several sink components, for each of which an individual S-QoS can be requested. In our example depicted in Figure 1, S-QoS would be specified at *Dsp1* and *Dsp2*.

The client is responsible to map the C-QoS specified by the end user to the S-QoS at the session level. Of course, for a given type of S-QoS, various types of C-QoS and mappings are conceivable. For example, the C-QoS value range of high, medium, low" can be mapped in many ways to the value ranges of the media parameters of an "uncompressed_video" stream. Obviously, this mapping depends on the class of application and end users.

At the transport level, T-QoS is specified in a media independent fashion. QoS at this level (and the network level) has been and still is subject to intensive research. Various QoS models have been proposed (e.g. see [1]), defining parameters, such as packet size, packet rate, burst size, delay, and jitter.

At all levels, we allow value ranges to be specified for QoS parameters. For a value range we assume the semantics as motivated in [1]: a max value denoting the desired QoS value and a min value denoting the worst acceptable one. The concept of value ranges is supported by the majority of real-time transport (and network) protocols (e.g., see Tenet Suite [1], or ST-II [2], RSVP [3] or XTPX [4]).

During session establishment, flowspecs are used to convey QoS information at the session and transport level. At the transport level, a flowspec carries media-independent parameters (e.g., see the ST-II flowspec [2]) and is communicated along the selected network path. At the session level, a flowspec (SFS) contains media specific and generic parameters and is passed along the flow graph structure, where both links and components are involved in QoS negotiation. The way how the flow specs are passed is defined by the respective negotiation and resource reservation protocol. At the transport level, one may apply one of the protocols mentioned above. For the session level, we propose the XNRP protocol presented in this paper.

In the CINEMA system, the mapping from media-specific QoS parameters to the T-QoS parameters is done within the link objects [15]. The media parameter values received in an SFS are mapped to the T-QoS parameter values of the transport mechanism encapsulated by the link. Whenever a link is requested to reserve resources for a data stream characterized by a set of media parameter values (e.g., values for frame rate and frame size), it derives the corresponding T-QoS values (e.g. packet size and rate values) and delivers them to the transport mechanism when issuing the connect request. Note that encapsulating this mapping in links makes the system rather independent of the underlying transport mechanism(s).

4 Refinement of the Application Model

In the following, we will refine the application model described in Section 2 by introducing the concepts of format constraints, stream relations and variable filters.

As ports are typed, only a certain type of stream can be communicated via a given port. However, the instances of a particular stream type may significantly differ in their media parameter values. It cannot be assumed that components support the entire value range for each media parameter. For example, an uncompressed_video filter may only support an 8 bit color depth and picture sizes up to a value of 1024x768. To express the capabilities of components with regard to the stream formats they are able to consume and/or produce, we introduce so-called ***format constraints***[1]. Each port is associated with a (possibly empty) set of format constraints. For a given port the associated format constraint set may contain a constraint for each media parameter defined for the port type. A constraint for a media parameter indicates the value range that can be supported by the component at that port.

In addition to format constraints, so-called ***stream relations*** have to be considered by the negotiation process. For example, an audio mixer mixing several audio streams usually requires the same quality to be delivered at all input ports, while a video mixer component mixing two videos into a picture-in-picture video, may require a fixed ratio of the picture sizes of the two video streams received at its input ports. In addition, the stream generated by the mixer is usually related to the stream received at one of the input ports. For instance, the video mixer may generate a video with the same frame size as available at the input port designated to receive the “larger” video. Similar dependencies may exist for filters. While there might be filters that do not change a stream's media parameters at all, others may reduce them by a certain factor, e.g., a video filter halving the picture size of the received video stream. Of course, those dependencies impact media parameter negotiation and hence must be provided by component designers in form of stream relations.

For mixer components, mixing relations are defined between pairs of input ports. They have the form: *param*: *@inport_i* = $f_{i,j}$**@inport_j*, meaning that the value of media parameter param at input port *inport_i* is to be $f_{i,j}$ times the value of this parameter at *inport_j*. To describe the relation between an output port and an input port of a mixer, a similar relation, termed filter relation, is defined: *@outport_i* = $f_{i,j}$**@inport_j*. This type of relation can be used to model the dependencies in our mixer examples above. A similar relation is defined for a fixed filter changing media parameter *param*: *@outport* = *F*@inport*, where *F* denotes the filter factor.

Obviously, the stream relations defined at a component not only affect the media parameters of this component's input and output streams but may impact the entire flow graph. Consider for instance the topology depicted in Figure 1. Assuming that at the input ports of the mixer the same video frame size is expected and that neither mixer nor *F1* and *F2* change the frame size, it is clear that session establishment requires the same frame size value at all ports in the flow graph. Hence, any combinations of S-QoS requests and format constraints violating this requirement causes the negotiation process to fail.

Multicast links cause similar dependencies. For instance, if in Figure 1 the S-QoS specs for *Dsp1* and *Dsp2* request frame size value ranges that do not overlap, negotiation will fail due to the multicast dependency imposed by the link leading to *F1* and *F2*. Even if the ranges overlap, the resulting frame size for both *Dsp1* and *Dsp2* would (at best) correspond to the intersection of the two ranges, i.e. the more modest request would govern the S-QoS for both sinks. Similarly, a resource shortage in one part of the flow graph may affect S-QoS in other parts due to the dependencies described above. For example, a resource shortage of the link leading to Dsp1, would impose an effect on *Dsp2* as well, even if the path from *Mix* to *Dsp2* would not suffer from any resource shortage.

In order to avoid such situations, we consider two classes of filters: fixed and ***variable filters***. Fixed filters correspond to the filters (with the relation type) described above. Variable filters are introduced to limit the scope of the dependencies imposed by multicast links and stream relations. A variable filter is supposed to be able to reduce a media parameter (e.g. frame size) value supported at its input port to any lower value supported at its output port, i.e., the stream relation is of the form: *param*: *@inport* >= *@outport*. The main difference to a fixed filter is that the filter factor is not determined in advance (i.e., by design), but is determined during negotiation. As will be seen later, the negotiation process determines and adjusts the filter factors of all variable filters in the flow graph, based on the format constraints, the S-QoS requests and the availability of resources at negotiation time.

Assume in our example of Figure 1 that *F1* and *F2* are variable filters for reducing frame_size. Now the negotiation process succeeds even if the frame_size ranges requested for *Dsp1* and *Dsp2* do not overlap because *F1* and *F2* can reduce the frame_size provided by *Mix* individually. This usage of variable filters makes it possible to provide for *Dsp1* and *Dsp2* (at best) their highest requested frame_size. Similarly, resource shortages of the output link of

1. A similar concept has been introduced by IMA [10]

F1, for instance, will not have any effect on the S-QoS at *Dsp2* since *F1* is able to reduce the incoming frame size down to the frame size for which the bandwidth available on the link to *Dsp1* suffices.

5 The XNRP Protocol

XNRP (eXtended Negotiation and Resource Reservation Protocol) was designed to support session establishment for arbitrarily complex flow graphs, composed out of sink and source components, mixers, fixed and variable filters interconnected via unicast and multicast links. When establishing a session, XNRP takes into account S-QoS specs, the various format constraints as well as the resources available at set-up time.

For the sake of simplicity, we make the following assumptions for the subsequent discussion. First, we will focus discussion on the negotiation of media-specific parameters. Generic parameters, such as delay and jitter are treated in a similar way as in existing network-level reservation protocols. For example, the delay introduced by links and components is accumulated during the second phase of our protocol and adapted during relaxation in phase 3.

Second, when specifying XNRP below, we will relate to one media parameter only. Third, we constrain the number of ports for the components introduced in Section 2. Source and sink components are assumed to have just one output resp. input port, while filters are assumed to have one input and one output port. Mixing components are supposed to have several input ports and one output port (with an equality stream relation to a specific input port). Much more general classes can be supported. As a plausibility argument take the possibility to model more complex components out of components defined. For instance, a multicaster component (with one input port and several output ports) can be modelled as a sequence of a filter connected via a multicast link to a set of filters.

Protocol Overview

XNRP proceeds in three phases. In each phase, Session Flow Specs (SFSs) are “swept” through the whole flow-graph, either in source-to-sink (i.e. downstream) direction or in sink-to-source (i.e. upstream) direction. During each phase, SFSs are delivered to the components and links being passed along the flow graph. The delivered SFS is interpreted, possibly modified and then returned to XNRP for further passing.

When SFSs are passed in a downstream direction, components and links receive an SFS for each input port and return a (possibly modified) SFS for each output port. In case of upstream passing, an SFS is received for each output port, and one SFS is returned for each input port. (We assume that links “share” ports with components.)

In the ***first phase*** of XNRP, information concerning the functional capabilities of components is propagated in a downstream direction. Receiving this information, sinks know which media parameter ranges are functionally supported by the upstream components. Phase 1 starts at the sources of the flow graph with the generation of the initial SFSs, indicating source format constraints. XNRP passes the SFS to subsequent (downstream) components. A component receiving SFS, intersects the included (media) parameter ranges with its format constraints and returns the possibly modified SFS to XNRP for further passing. Upon reaching a sink, XNRP passes the SFS and the specified S-QoS to the sink. From this sink’s point of view, at this point, the second phase starts.[1]

The ***second phase***, the reservation phase, serves to reserve resources for components and links[2]. The phase starts at sinks, where reservation is done according to received SFS, the local format constraints and the specified S-QoS. In case of a resource shortage, reservation is done for the best possible parameter value. In the case of shortages, the SFS range is reduced correspondingly and returned to XNRP. XNRP passes the SFSs successively to the links and components in an upstream direction. Each of them attempts to reserve resources for the upper range boundary indicated in the received SFS. In case of resource shortages, they reserve for the best possible value and return an SFS with a reduced range, observing their respective stream relations. From a source’s point of view, phase 2 ends after it has reserved resources and returned the possibly modified SFS to XNRP. Of course, if some component or link is unable to reserve enough resources (i.e., to keep the media parameter values within the acceptable ranges), XNRP terminates.

1. Note that during the first phase, only format constraints are taken into account (and no resource reservation is performed). Links are omitted in this phase, since they do not have such constraints.

2. A component interacts with resource managers (e.g. for CPU, buffer) for obtaining required resources. A link employs the encapsulated transport reservation protocol to set-up a required connection.

As will be shown below, observing stream relations only inside components is not sufficient. In order to capture flow graph-wide dependencies, in the second phase XNRP additionally collects stream relations and passes them to a central entity, called QoS orchestrator. The QoS orchestrator uses this information to compute the optimal value setting of variable filters and sources included in the flow graph. The computation results are returned to the variable filters and sources, which exploit this information during phase 3.

In the ***third phase***, the relaxation phase, SFSs are propagated from sinks to sources incorporating results returned by the orchestrator. They are used by components and links to derive their reservation relaxation. Upon reaching all source components, XNRP has installed at all component ports the best possible media parameter values. At this point, session establishment ends.

It is important to point out that the orchestrator can be a different entity for each session and can be located on a node hosting essential components of the session anyway. Thus, this architecture is in line with the fate-sharing principle introduced in [16].

Need for Central Orchestration

Why do we need a central entity for exploiting stream relations at all ? Consider the flow graph depicted in Figure 4 and assume that the (one) media parameter, frame_size is to be negotiated. Assume further both *Mix1* and *Mix2* are associated with a mixing relation that requires the same frame_size at the mixer input ports. Finally, it is assumed that all components and links have abundant resources and no format constraints are given for component ports. Nevertheless, the flow graph implies global dependencies. For instance, all three sources have to provide the maximum of the frame_sizes requested at *Dsp1* and *Dsp2*. This is due to the stream relation introduced at *Mix1* and *Mix2*. If, for instance, the biggest frame_size is specified for *Dsp1*, this value has to be taken into account at *V3* as well, in order to provide the appropriate frame_size and to reserve properly.

An intuitive approach is to propagate the QoS information following the structure of the flow graph. In the second phase, starting at sinks, the QoS information specified at *Dsp1* can be propagated only to *VF1*, *Mix1*, *V1* and *V2*. In the third phase, starting at the sources, this information will reach *Mix2*, *VF2* and *Dsp2*, and only after the third phase, it could be available at *V3*. Therefore, with this approach, at least four phases are needed for the given flow graph. Obviously, the number of required phases depends on the structure of the underlying flow graph. For example, additional phases would be needed if the flow graph of Figure 4 were extended by adding more sources and mixers, coupling pairwise adjacent sources. In consequence, if this approach is applied on arbitrarily structured flow graphs, no upper bound on the number of phases required for session set-up can be given.

XNRP adopts a different approach. Dependency information that can be derived from stream relations is collected in the second phase and sent to the orchestrator. With this information the orchestrator has a global view on the dependencies in the flow graph, and hence is able compute the optimal settings for sources and variable filters. Below we will describe the basic principles of this approach.

Variable filters and sources have variable output within the limitations given by their format constraints. This fact is indicated in XNRP by means of so-called filter variables. Each source and each variable filter generates a filter variable, which is included in the SFS and propagated downstream in phase one. A filter variable is propagated until it hits another variable filter or it reaches a sink. At a mixer, one variable reaches each input port, but only one (arbitrary) is propagated. In Fig. 4, *V1*, *V2*, *V3*, *VF1* and *VF2* are assumed to generate filter variables *v1*, *v2*, *v3*, *vf1* and *vf2*, respectively. The illustration indicates for each input port the filter variable received at this port.

For each source, mixer and variable filter XNRP interacts with the orchestrator. For each mixing relation *param*: *@inport2*=$f_{2,1}$**inport1* defined at a mixer the following equation is sent to the orchestrator: *param*: $v_i = f_{i,j} * v_j$, where it is assumed that filter variables v_i and v_j were received at *inport2* and *inport1*, respectively. In our example, the equations *v1=v2* and *v2=v3* are sent for *Mix1* and *Mix2*, respectively. (Since we consider only one media parameter, we have omitted the *param* specifier).

For a variable filter reducing parameter param XNRP sends an inequation param: $v_i >= v_j$ to the orchestrator, where it is assumed that v_i was received at the filter's input port and v_j is the filter variable generated by this filter. Moreover, XNRP includes the "guaranteed" range of v_j in the message sent to the orchestrator. Since this range is determined in phase 2 after the variable filter and all downstream components and links have already reserved resources for this range, it is called "guaranteed". In our example, *VF1* and *VF2* would contribute inequations *v1>=vf1* and *v2>=vf2* and a guaranteed range for *vf1* and *vf2*, respectively. As sources are not associated with stream relations, they only contribute guaranteed ranges for their filter variables. In our example, XNRP delivers the guaranteed ranges for *v1*, *v2* and *v3* to the orchestrator.

After having received all information[1], the orchestrator starts to solve the system of received in/equations taking into account the guaranteed variable ranges. As a result, it will compute an optimal value for each filter variable. Optimal means that - given all format constraints, S-QoS requests and the resource availability - the best possible S-QoS is achieved at every sink. It additionally means that the variable values are determined in a way that filtering (i.e. QoS reduction) is done as close as possible to the sources.

The orchestrator delivers the computed values to the variable filters and sources of the flow graph. During the third phase of XNRP, this information is used by the sources to determine the media parameters of their output, while it is used by variable filters to determine their actual filter factor.

The advantage of the XNRP approach is obvious: Independent of the structure of the flow graph only three phases are needed for session establishment. The price is the additional communication overhead with the orchestrator function: two extra messages for each variable filter, source and mixer in the flow graph.

Structure of SFS, OReq and OResp

The SFS employed by XNRP consists of a part carrying information for media parameters and a second part foreseen for generic parameters (delay, etc.).

```
SFS{  MediaParameterPart(
          record_for_param1(
              varId                // variable representing the closest upstream source or variable filter
              factor               // proportionality factor relating varId to current value range
              valR)                // value range for parameter at current port: min, max values
          ...
          record_for_paramN(...) )

      GenericParameterPart(...) }
```

The structure for an OReq allows to include a list of stream relations and one value range per media parameter:

```
OReq{record_for_param1(
         varId                         // Id of variable for which range is given
         valR                          // value range: min, max values
         s_rel_1(
             type                      // equality, inequality
             var1                      // Id of first variable in stream relation
             var2                      // Id of second variable in stream relation
             factor )                  // relation factor (e.g. var1 * factor >= var2)
         ...
         s_rel_M(...))
    ...
     record_for_paramN(...)}
```

Finally, an OResp contains a value for media parameters at a specific source or variable filter output port:

```
OResp{record_for_param1(
         varId                         // Id of variable for which value is given
         value )
    ... record_for_paramN(...)}
```

1. To decide that it has received all information, the orchestrator at least must know the number of sources, mixers and variable filters included in a particular flow graph. This information could be provided by a management function maintaining information about applications. However, it could be also a parameter of the session, which is delivered to the orchestrator when set-up is initiated.

Protocol specification

NRP is performed by a set of protocol agents (PA). Figure 3 shows a realization with one PA per component (C) and link (L), where PAs are interconnected according to the component topology of the application.

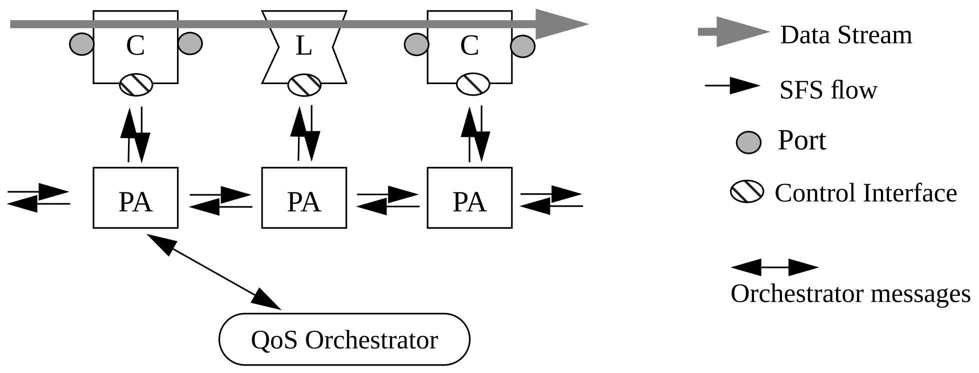

Figure 3 : Implementation Architecture

An important feature of XNRP is that it allows components and links to take a local view to negotiation. They only have to support a few methods (see below) which are invoked by XNRP (PAs) during negotiation. A component or link is only required to interact with its associated PA. XNRP provides for correct SFS passing between components and links. This passing is done towards components and links via component and link method invocations. In addition, interconnected XNRP PAs exchange SFS according to the direction of the protocol phase. In the second phase PAs of source components, mixers and variable filters send orchestration requests to the orchestrator. The orchestrator return responses to the corresponding PAs which are used in the third phase.

XNRP requires each component and each link to provide a generic interface for negotiation consisting of a set of methods: *Prepare_Reservation* (invoked during phase 1), *Reserve* (invoked during phase 2), and *Relax* (invoked during phase 3). A fourth method *Free* has to be offered to indicate to a component or link that a reservation session is to be terminated. In the following, we focus on component and link behavior in each phase.

Phase 1 (Prepare Reservation)

In this phase, each component receives at once a set of SFS_{in} (one for each input port) and calculates an SFS_{out} for its output port. First, for each input port, the SFS_{in} is intersected with the port's format constraint. Second, if mixing relations exist between input ports, they are used to intersect all SFS_{in}. Third, the filter relation between the output port and the (the selected) input port is applied, to calculate SFS_{out}. Depending on the component type, a fixed relation is used with a fixed factor f (for fixed filters and mixers) or a variable relation (for variable filters).

A *source component* generates an SFS_{out} indicating its format constraints: SFS_{out}(sourceId, factor(1), valR(max..min)), where sourceId is the variable representing the stream at the source port, factor indicates the initial proportionality factor of 1, while valR gives the supportable range (format constraint) at the source port.

A *fixed filter* receives an SFS_{in}(varId, factor, valR(max..min)) indicating the requested value range at its input port and its stream relation to the upstream source or variable filter denoted by varId. This relation is described by the proportionality factor. The filter calculates for its output port SFS_{out}(varId, f * factor, valR(f*max..f*min)), where f indicates the filter factor used in the filter relation.

A *mixer* receives for each input port i an $SFS_{in,i}$(varId$_i$, factor$_i$, valR(max$_i$...min$_i$)). The mixer produces an SFS_{out}(varId$_k$, factor$_k$, valR(max..min)). The mixer relates the stream at its output port to the varId received at the input port k to which it has a defined relation. As mentioned, for a mixer this relation is set to equality, so the corresponding factor is not changed. Thus, valR in SFS_{out} reflects the value range obtained for inport k. This is obtained by intersection of all received value ranges (observing mixing relations defined between k and the other input ports), i.e. max in SFS_{out} is given by Min(max$_i$/f$_{i,k}$) and min is given by Max(min$_i$/f$_{i,k}$). In addition, a mixer stores for each input port the received $SFS_{in,i}$, (for later use during phase 2).

A *variable filter* receives an SFS_{in}(varId, factor, valR(max..min)) and generates an SFS_{out}(own_varId, factor(1), valR(max..own_min)). The filter inserts its own varId, given that downstream components can relate in a fixed way their port parameter now to the output port of this variable filter, but not to other upstream components (since a variable filter is in between). Obeying its variable relation (param: @inport >= @outport), the filter inserts in valR the lower limit of its output format constraint as min, while taking min(received max, upper bound of output format constraint) as the new max. In addition, the filter stores the SFS received at the input port (for phase 2).

A *sink* component receives SFS_{in}(varId, factor, valR(max...min)) and the S-QoS(valR(max..min)) requirement of the client. The generation of an SFS_{out} is part of the second phase.

Links are not invoked by XNRP during the first phase.

Phase 2 (Resource Reservation)

In the reservation phase each component is invoked with an SFS for its output port. Each component tries to reserve resources according to received SFS. If resources do not suffice, the upper limit of the SFS range is reduced to a value (in the range) for which reservation is possible. Based on reservation, an SFS is derived for each input port of the component, observing the stream relation(s) of the component defined between input and output ports. In this phase, in addition, stream relations and value ranges are derived for filter variables and indicated to XNRP.

A link is invoked with one SFS for each connected (receiving) input port. Based on this, the link invokes an underlying transport system with calculated T-QoS values for setting up a (unicast or multicast) connection. T-QoS values accepted by the transport system are calculated into the media representation of an SFS for the sending side of the link (see [15] for details). This SFS is handed back to XNRP.

In the second phase, only some parts of SFS are relevant. We omit mentioning the other parts, where unnecessary.

A *sink* component reserves resources according to intersected ranges of received SFS_{in} (of phase 1), S-QoS and its own format constraints. It generates a SFS_{in}(valR(val...min)) with val reflecting the value for which was reserved.

A *fixed filter* receives at its output port an SFS_{out}(valR(max...min)) and tries to reserve accordingly. Assuming that it could reserve for value val with respect to its output port, the SFS to be generated for its input port has to be SFS_{in}(valR(val/f...min/f)) where f denotes the filter factor.

A *mixer* receives SFS_{out}(valR(max...min)) for its output port and tries to reserve accordingly. Assuming that it could reserve for value val with respect to its output port, the SFS to be generated for the input ports are as follows. For the input port k for which the equality relation with the output port was defined, the SFS is derived: $SFS_{in,k}$(valR(val...min)). The SFS for all other input ports are derived from this SFS using the corresponding mixing relations between input ports: $SFS_{in,i}$(valR(val*$f_{i,k}$..min*$f_{i,k}$) with $f_{i,k}$ denoting mixing factors.

In addition, a mixer uses its mixing relations to derive relations between the filter variables received (and stored) in the first phase at the input ports. Assuming that at port k ($varId_k$, $factor_k$) has been stored and at another input port i ($varId_i$, $factor_i$), the mixer derives a relation s_rel_i(type(equal), $varId_k$, $varId_i$, new_factor). new_factor is derived from the mixing relation between inports i and k: $varId_i$ *$factor_i$ = $factor_{i,k}$*$varId_k$ *$factor_k$ and is given as $factor_k$*$factor_{i,k}$/$factor_i$. Both the $SFS_{in,i}$ and the s_rel_i are relayed to XNRP. The s_rel_i are relayed to the orchestrator via an OReq(list_of_s_rel_i). No value ranges are relayed for the variables of s_rel_i.

A *variable filter* receives an SFS_{out}(valR(max...min)) for its output port. It tries to reserve according to this range[1]. Assuming the filter could reserve for value val with respect to its output port, the SFS generated for its input port is SFS_{in}(valR(new_max..new_min)), where new_max is the max value stored for the input port in the first phase. new_min is given as Max(min value of input port stored in phase 1, min received in phase 2 for the output port). new_min has to be calculated in this way, since any value at the input port has to have a possible value at the output port (given that a variable filter can only reduce values).

In addition, a variable filter relays a stream relation coupling the filter variable received in phase 1 to its own (output port) filter variable. Assuming that at the input port ($varId_i$, $factor_i$) has been stored in phase 1, the following relation is compiled: s_rel(type(ineq), $varId_i$, own_varId, factor). factor is derived from the filter's stream relation between input and output port $varId_i$ * $factor_i$ >= own_varId and is given as $factor_i$. Both the SFS_{in} and the s_rel

1. A variable filter, depending on its specific implementation, may reserve resources either to received SFS for the output port or to the SFS received and stored in the first phase for the input port.We consider here only variable filters which are "output driven" concerning resource demand,. A filter selecting parts (e.g. pixels) of received data unit is an example for this. We assume that the buffer space required by incoming data is reserved by the link prior to the filter.

are relayed to XNRP. In addition, a variable filter indicates the “guaranteed” value range for its output port for which resource reservation was possible: (own_varId, valR(val..min)). XNRP forms a corresponding OReq(own_varId, valR(val..min), s_rel( )) which it relays to the orchestrator.

A *source component* receives an SFS_{out}(valR(max...min)) for its output port. Assuming that it can reserve for value val with respect to its output port, it compiles SFS_{out}(own_varId, valR(val...min)) as response to XNRP. XNRP forms a corresponding OReq(own_varId, valR(val..min)) which it relays to the orchestrator.

The QoS orchestrator solves the system of equalities and inequalites for the filter variables received in OReq. For each variable, it returns in an OResp(varId, value) the final value to the respective PA (for use in phase 3).

Phase 3 (Resource Relaxation)

In this phase SFS are propagated containing only one value in valR. The other parts are omitted below.

A *source component* XNRP receives an SFS(valR(value)), relaxes reservation, if reservation in phase 2 covered a higher value, and returns the SFS unchanged to XNRP.

A *fixed filter* receives SFS_{in}(valR(value)), possibly relaxes reservation and returns SFS_{out}(valR(f*value)) to XNRP, where f denotes the filter factor.

A *mixer* receives an $SFS_{in,i}$(valR($value_i$) for each input port, possibly relaxes reservation, and returns SF-S_{out}(valR($value_k$)) to XNRP, where k denotes the input port to which the output port has its equality relation.

A *variable filter* is given by XNRP at once two SFS, one for its input and one for its output port. The SF-S_{in}(valR($value_{in}$)) is received from upstream direction and contains the value to be installed at the input port. The SFS_{out}(valR($value_{out}$)) is relayed by the XNRP PA to the component, after receiving a corresponding OResp message. It installs the final value at the filter output port. The filter possibly relaxes reservation and responds with an unchanged SFS_{out}(valR($value_{out}$)) to indicate the end of relaxation. Note, by installing input and output value, the filter fixes the previously variable relation between its input and output port.

A *sink component* receives SFS_{in}(valR(value)), possibly relaxes reservation, and returns the SFS to XNRP. Upon receiving all sink responses, a session is set up and communication can take place.

A *link* is passed an SFS_{in}(valR(value)) for its sending side. It calculates corresponding T-QoS values, and initiates relaxation for the encapsulated transport connection. It returns to XNRP an SFS for each receiving side of the link.

Negotiation Example

Figure 4 depicts a video application scenario with two participants. Each participant can view two videos at once in form of a picture-in-picture mixture. Participant 1 can control Mix1 to select which of the two videos V1, V2 shall be the large one. P2 can control Mix2 similarly for videos V2 and V3. Variable filters VF1 and VF2 allow for individual provision of S-QoS at Dsp1 and Dsp2.We use here frame size as the media parameter to be negotiated. In order to keep description simple, we leave out the links (unless they are explicitly mentioned), assuming that they can reserve resources as required. We assume that all component ports have a format constraint of (1024x768...120x90) pixels. The client specifies at Dsp1 S-QoS(800x600...480x360), and at Dsp2 S-QoS(480x360...240x180). We assume that Mix1 requires the same frame size at its input ports. In addition, the frame size of Mix1’s output port shall be related to the upper input port (dotted line in the figure). Analogous assumptions are made for Mix2.

XNRP starts at sources with SFS(v1, 1, 1024x768...120x90) generated by V1, SFS(v2, 1, 1024x768...120x90) generated by V2 and SFS(v3, 1, 1024x768...120x90) generated by V3[1]. The SFSs from V1 and V2 are propagated to Mix1, which generates SFS(v1, 1, 1024x768...120x90). The SFS from V2 and V3 are passed to Mix2 which generates SFS(v2, 1, 1024x768...120x90). VF1 is passed the SFS from Mix1, upon which it generates SFS(vf1, 1, 1024x768...120x90). VF2 is passed the SFS coming from Mix2 and generates SFS(vf2, 1, 1024x768...120x90). Dsp1 is passed the SFS from VF1, Dsp2 the SFS from VF2. In this phase, received SFS are stored by Mix1, Mix2 VF1 and VF2. The phase ends upon reaching Dsp1 and Dsp2.

1. We use this shorter notation SFS(v1, 1, 1024x768...120x90) instead of SFS(varId(v1), factor(1), valR(1024x768...120x90)).

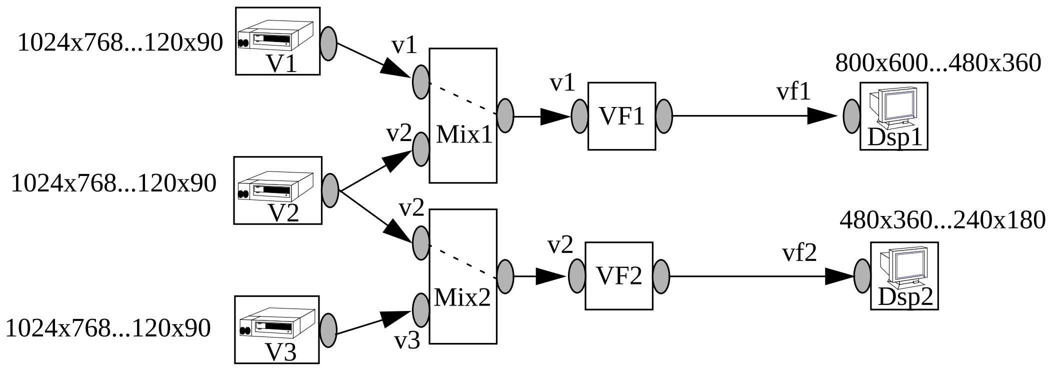

Figure 4 : Video Application Scenario

The second phase proceeds from the sinks towards the sources. The SFS range received by a sink is intersected with the respective S-QoS request and the own format constraint. For Dsp1 this yields valR((800x600...480x360), for Dsp2 valR(480x360...240x180). Assuming that reservation is successful for best values in both cases, Dsp1 returns SFS(valR((800x600...480x360)), while Dsp2 returns valR(480x360...240x180). Assuming that the link between VF1 and Dsp1 could only reserve for (640x480) due to limited bandwidth, the SFS passed to the link is changed to SFS(640x480...480x360). This SFS is returned by the link and passed to variable filter VF1.

VF1 can reserve for the whole received range. It generates an SFS(1024x768...480x360) for its input port. It compiles a stream relation between the variable received in phase 1 (v1) and its own variable (vf1): s_rel(ineq, v1, vf1, 1). SFS, the stream relation and the “guaranteed” value range for vf1 are passed to XNRP. XNRP sends an OReq(vf1, 640x480...480x360, s_rel) to the QoS Orchestrator. VF2 is passed the SFS(valR(480x360...240x180)) from Dsp2. Assuming it can reserve for the best value, it generates an SFS(1024x768...240x180) for its input port. In addition s_rel(ineq, v2, vf2, 1) is derived. SFS, s_rel and the range for vf2 are passed to XNRP. XNRP sends an OReq(vf2, 480x360...240x180), s_rel).

Mix1 receives the SFS from VF1 and shall be able to reserve for the best value. It generates two similar SFS(1024x768...480x360) for its input ports. In addition, it compiles an s_rel(eq, v1, v2, 1). Both SFS and s_rel are passed to XNRP. XNRP sends an OReq(s_rel) to the orchestrator. Mix2 is passed the SFS(1024x768...240x180). It shall be able to reserve for the best value. It generates two similar SFS(1024x768...240x180) for each of its input ports. In addition, it compiles an s_rel(eq, v2, v3, 1). Both SFS and s_rel are passed to XNRP. XNRP sends an OReq(s_rel) to the orchestrator.

V1 receives the SFS from Mix1. It shall be able to reserve for the best value. It return SFS(v1, 1024x768...480x360) to XNRP, which compiles an OReq(v1, 1024x768...480x360). The multicast link between V2, Mix1 and Mix2 intersects the ranges of the AFS received for Mix1 and Mix2. It returns SFS(1024x768....480x360) which is passed to V2. V2 shall be able to reserve for the best value. It returns SFS(v2, 1024x768....480x360) which is used by XNRP to send an OReq(v2, 1024x768....480x360). V3 is passed the SFS(1024x768...240x180) from Mix2. It shall be able to reserve for the best value. It returns SFS(v3, 1024x768...240x180) which is used by XNRP to send an OReq(v3, 1024x768...240x180).

During the second phase the orchestrator has received the following (in)equalities, variables and (guaranteed) value ranges: v1>=vf1, v2>=vf2, v1=v2, v2=v3, (1024x768...480x360) for v1 and v2, (1024x768...240x180) for v3, (640x480...480x360) for vf1 and (480x360...240x180) for vf2. The orchestrator solves this system, such that all relations are observed. For vf1 and vf2 it finds the best providable values: 640x480 resp. 480x360. For v1, v2, v3 it finds the lowest values which ensure best values for vf1 and vf2. In this case, v1, v2, v3 are set to 640x480. These values are passed to XNRP in corresponding OResp messages and used in the third phase.

In the third phase, V1 is passed SFS(640x480) for v1. It relaxes reservation according to that frame size and returns the SFS unchanged. V2 and V3 act similarly. Mix1 is passed an SFS(640x480) for each of its input port. It relaxes reservation and returns the SFS unchanged. Mix2 acts similarly.

VF1 receives from XNRP two SFS: SFS(640x480) for its input port and SFS(640x480) for its output port. The filter deduces that no processing is required at all, and relinguishes resources acquired during phase 1. It returns SFS(640x480). VF2 receives SFS(640x480) for its input port, and SFS(480x360) for its output port. It deduces that a corresponding frame size reduction is required and fixes its internal processing accordingly. It returns SFS(480x360).

Dsp1 is passed SFS(640x480) and relaxes reservation accordingly. Dsp2 is passed SFS(480x360). It does not have to relax reservation. At this point, session establishement is finished.

QoS Orchestrator

The orchestrator receives a set of OReq containing equality and inequality relations between filter variables. The number of equations equals the number of mixing relations (m), while the number of inequalities is bounded by the number of sources plus variable filters (n). In addition, for each variable, a value range is received indicating possible “guaranteed” variable values. The algorithm used to solve the system of relations is performed in two steps. The first step solves the system, determining for each variable filter the highest possible value. The second parts leaves values for variables unchanged which are related in a fixed (proportional) way to sinks, while minimising all other variable values. We give below an overview of the algorithm performed, and refer to [17] for a more formal and detailed description.

The first step is performed in several iterations. Each iteration starts with a value for each variable. Initially, the upper bounds of the received ranges are used. Each relation which is not satisfied by the current two variable values, changes one or both values to satisfy the relation. Values are always reduced, and reduction is done stepwise, i.e. from one value to the next lower one. When reductions have rendered the relation satisfied, the new (reduced) values are inserted for the related variables. All relations are considered in this manner in each iteration (the sequence of relation consideration can be arbitrary).

Once an iteration did not change any variable value, the first step is finished, since the variable values satisfy all relations. If at least one change occurred, the next iteration is triggered. Let k denote the number of values which are maximally possible for a media parameter, i.e. for any of the n considered variables. Correspondingly, the maximal number of value reductions is bounded by (k-1)*n. After this number of reductions, the smallest media values are reached for all variables. Any further reduction indicates non-solvability of the relation system. Since per iteration, at least one of these variables is reduced, at whole maximally k*n iterations can occur. Since per iteration (m+n) relations are considered (touched), the worst complexity of this step is of order O((m+n)*n*k).

The second step distinguishes the variables to which sink output ports are related in a fixed (proportional) way. In [17] it is shown that these variables can be identified in the relation system (basically the variables are considered which do not have a greater-than relation to other variables)[1]. The values for these variables are retained from the first step. For all other variables, the minimal values from their respective value ranges are inserted initially and adjustions are applied, until all stream relations are satisfied again.

The adjustions increase variable values. Otherwise the procedure is the same as for the reductions of the first step. In particular, the computation complexity is again bounded by O((m+n)*n*k). Note that if the first step yielded a solution, the second step cannot miss it, i.e. the second step renders a solution with lower or equal values to the ones found in the first step. Also note, that the first step already delivers a possible solution fixing (implicitly) the final values for sink ports. The second step merely serves to reduce variable values as early (i.e. close to sources) as possible in the flowgraph, in order to save resources. This reduction is done such that the sink port values implied in the first step are not changed.

For the example in Figure 1, the first step starts with values: 1024x768 for v1, v2, v3, 640x480 for vf1 and 480x360 for vf2. Given that all relations are already satisifed, the first step ends here. The second step leaves values for vf1 and vf2 unchanged, since they are related in a fixed way to sink ports Dsp1 and Dsp2. Starting values for the second step for the other variables are: 480x360 for v1, v2 and 240x180 for v3. Applying the stream relations iteratively, yields 640x480 for v1, v2 and v3. All final values are returned to corresponding XNRP PAs.

The QoS Orchestrator has been implemented and evaluated for performance. For a flowgraph containing m mixers, and n sources resp. variable filters the following (worst-case) results have been obtained for computation time on a SunSparc10 machine running at 50 MHz:

1. One may assume, that XNRP indicates in OReq (in a yes/no flag) whether or not a contained variable is related in a fixed way to a sink port.

(m=2, n=2, k=5): computation time = 0.5 ms
(m=16, n=16, k=5): computation time = 5 ms
(m=100, n=100, k=5): computation time = 180 ms.

These results indicate that the computation time is low, even for quite large flowgraphs. In consequence, for any flowgraph of such or lower size, the QoS orchestrator will not add undue overhead to negotiation.

6 Conclusion

The *Cinema* system allows a client to set up and control distributed multimedia applications composed of components interconnected via links. A client is offered three service classes: configuration management, session management and synchronisation management. Session management allows to instantiate an application according to QoS specified by a client.

Session set-up is based on a QoS architecture distinguishing between client, session and transport level QoS. Session level negotiation is performed by the *Cinema* session management. Its purpose is to orchestrate between client A-QoS requests given in form of parameter value ranges and constraints imposed on an application: format constraints at component ports, resource availability for components and links as well as stream relations defined between component ports. This orchestration has to be performed at the session level, i.e. in terms of media parameters associated with component ports.

XNRP is a protocol performing negotiation and resource reservation at the session level. It was design to be applicable to arbitrary flowgraphs containing source components, sink components and intermediate ones, such as mixers, fixed and variable filters. Variable filters allow to decouple topology dependencies of a flowgraph. XNRP is performed in three phases during which flowgraph capabilities (format constraints) are signalled to sinks, resources are reserved, final media values are calculated based on reservations and resource reservation is relaxed according to calculated values.

In order to bound the required number of protocol phases, final media value calculation is done by a central QoS orchestrator. XNRP obtains stream relations and value ranges for filter variables during its reservation phase and relays these to the orchestrator. The orchestrator solves the system of relations optimally and returns final media values which are used by XNRP for reservation relaxation. Performance results obtained for an implemented QoS orchestrator show that required computation time is low, even for large flowgraphs. Overall negotiation overhead is not unduly increased by the orchestrator.

7 References

[1] D. Ferrari et al. Network Support for Multimedia: A Discussion of the Tenet Approach, *Computer Networks and ISDN Systems 26,* special issue on Multimedia Networking, 1994

[2] C. Topolcic. Experimental Internet Stream Protocol, Version 2 (ST-II). *RFC 1190*, 10 1990.

[3] L. Zhang et al. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, p. 8–18, 9 1993.

[4] B. Metzler et al. Specification of the Broadband Transport Protocol XTPX. *Technical University of Berlin, available via http://www.prz.tu-berlin.de/docs/html/prot/xtpx.html*, 2 1993.

[5] K. Nahrstedt, J. Smith. The QOS Broker. *IEEE Multimedia, Vol. 2, No. 1,* p. 53-67, Spring 1995.

[6] G. Dermler, W. Fiederer, I. Barth, K. Rothermel. A Negotiation and Resource Reservation Protocol (NRP) for Distributed Multimedia Applications. *IEEE International Conference on Multimedia Computing and Systems,* Hiroshima, Japan, 1996.

[7] G. Blair et al. An Integrated Platform and Computational Model for Open Distributed Multimedia Applications. In *3rd International Workshop on Network and Operating System Support for Digital Audio and Video,* p. 209–222, 11 1992.

[8] T. Käppner et al. Eine verteilte Entwicklungs- und Laufzeitumgebung für multimediale Anwendungen. In *Proceedings of KiVS'95,* p. 76–86, 1995.

[9] J. Magee et al. An Overview of the REX Software Architecture. *2nd IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 10 1990.

[10] HP Company and IBM Corporation and SunSoft Inc. *Multimedia System Services, Version 1.0, available via ftp from ibminet.awdpa.ibm.com*, 7 1993.

[11] K. Rothermel, I. Barth, T. Helbig. In *Architecture and Protocols for High-Speed Networks*, Chapter CINEMA - An Architecture for Distributed Multimedia Applications, p. 253–271. Kluwer Academic Publishers, 1994.

[12] I. Barth. Configuring Distributed Multimedia Applications Using CINEMA. *IEEE Workshop on Multimedia Software Development MMSD 96*, Berlin, Germany, 1996.

[13] K. Rothermel, T. Helbig. Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams. *IEEE Journal on Selected Areas in Communications - Synchronization Issues in Multimedia Communications,* 1996

[14] K. Rothermel, T. Helbig. An Adaptive Protocol for Synchronizing Media Streams. *ACM/Springer Multimedia Systems*, 1996

[15] K. Rothermel, G. Dermler, W. Fiederer. A communication infrastructure for multimedia applications. *European Conference on Networks and Optical Communications,* Heidelberg, Germany, 1996.

[16] D. Clark. The design philosophy of the DARPA internet protocols. In *Proc. of ACM SIGCOMM 92*, pages 14-26, Baltimore, Maryland, August 1992.

[17] G. Dermler, W. Fiederer, K.Rothermel. Negotiation and resource reservation for multimedia applications. Technical Report, (http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/Publications.html)