

A Group and Session Management System for Distributed Multimedia Applications

Erik Wilde, Pascal Freiburghaus, Daniel Koller, Bernhard Plattner

Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology (ETH Zürich)
CH – 8092 Zürich

Abstract. Distributed multimedia applications are very demanding with respect to support they require from the underlying group communication platform. In this paper, an approach is described which aims at providing group communication platform designers with a component which can be used for powerful group and session management functionality. This component, which can be integrated into group communication platforms, is part of a system called the group and session management system (GMS). The GMS model consists of GMS user agents, which are the components to be integrated into group communication platforms, and GMS system agents which are distributed directory agents providing the distributed database which the user agents access. Communication between these two types of agents is defined in two protocols, the GMS access protocol between user agents and system agents, and the GMS system protocol between system agents. GMS also defines a number of objects and relations which can be used to manage users, groups, and sessions on a very abstract level, thus providing both group communication platform designers and programmers of distributed multimedia application with a high-level description of group communications. This approach enables a truly integrated approach for collaborative applications, where all applications, even when using different group communication platforms, can share the same database about users, groups, and sessions. The paper also contains a short description of the ongoing implementation of GMS's components.

1 Introduction

Computer communications in the last years can, when looking from the level of a user of communication systems, be seen as focusing on two aspects, which are multimedia and multipoint communications. Both aspects can be approached in a number of different ways, ranging from very low-level, being concerned only with transport technology, to very abstract models, which take multimedia multipoint communications for granted and deal only with the design of applications and their interfaces. In general, various surveys [5, 16, 17, 19, 26] have shown that the need for communication platforms supporting multipoint communications is increasing. However, we feel that there is much more work being done in the field of multimedia than in the field of multiuser communication systems.

In this paper, we describe a system which is specifically designed to support distributed multimedia applications. Our approach is that of a component which can easily be integrated into group communication platforms to provide them with sophisticated group and session management functions. This design is motivated by the following observations:

- When considering group communications, the need for a common information base for current and potential group communication participants becomes very important. Only if the properties of a group communication are known to potential participants, it is possible for them to have enough knowledge to join the group communication. Thus, a directory for information related to group communications is necessary.
- Observations of research projects implementing distributed multimedia applications show that some of the functionality is repeatedly implemented in each application. Examples for this observation are the BERKOM Multimedia Collaboration Service (MMC) described by Altenhofen et al. [1] with its Conference Directory (CD), the CoDraft system described by Kirsche et al. [16] with its multi-party communication platform, and the Joint Viewing and Tele-Operation Service (JVTOS) described by Gutekunst et al. [7] with its Session Management Service (SMS). Consequently, it would also be useful to have a reusable software component which can be used to access the directory mentioned in the first item.

Another aspect of this issue is that, ironically, most of today's collaborative software is not able to collaborate with other collaborative software, because in each product a different model of collaboration (such as identification of users and groups, authorization, or creation of data connections) is used. We therefore also see our work as one step towards collaborative software which not only supports collaborative users, but also makes it easily possible to collaboratively use different products.

The design of the group and session management system primarily focuses on creating a model of group and session management which can be adopted for existing as well as new group communication frameworks and which provides application programmers (ie users of the group communication frameworks) with a powerful abstraction to handle group communications. In this paper we also describe an architecture which implements the group and session management model and which has been implemented at our lab. It is currently integrated into the Multipoint Communication Framework described by Bauer et al. [3], which will then serve as the first example of a group communication framework offering the above mentioned functionality.

The paper is structured as follows. Section 2 gives a short description of selected related work. In this section we will describe research as well as standardization activities. Section 3 describes the requirements which has been used when designing our model for group and session management. Section 4 then gives a description of the model as well as the architecture which we developed to implement the model. The main components here are the data model and the

two protocols which are used for data transfer within our system. In Section 5, we describe the implementation of our prototype. This section focuses on the two main building blocks of our architecture, which are two types of agents. Finally, Section 6 concludes the paper and gives some final remarks as well as some discussions of our plans for the future.

2 Related Work

In the Internet world, work is going on in the Multiparty Multimedia Session Control (MMUSIC) working group of the Internet Engineering Task Force (IETF). One result of this work is a description of the Internet multimedia conferencing architecture by Handley et al. [9], which is shown in Figure 1. The component of this architecture which is most relevant for our work is the session directory, which is based on the Session Description Protocol (SDP) v2 described by Handley and Jacobson [10]. However, since SDP can only be used for session advertisement, because it is only used for the distribution of announcements, an additional protocol is required which is used for specifically inviting users to sessions. This protocol is the Simple Conference Invitation Protocol (SCIP) and is described by Schulzrinne [22]. This splitting of session relevant information into two separate protocols has been caused by the development of the mbone (a good overview of the mbone is given by Eriksson [6]), which originally was only used for multicasting sessions with a simple session announcement protocol (SDP v1). Since a more powerful support of group communications also needs a way to identify users and groups of users, we believe that a new design than the one currently being in use for the mbone is necessary.

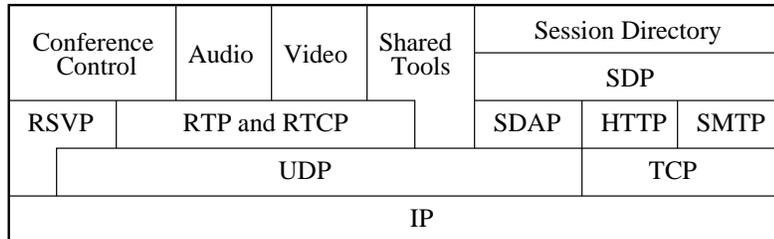


Fig. 1. Internet multimedia conferencing protocol stacks

CIO multi-peer communications as described by Henckel [11] is a very interesting concept in terms of functionality. The transport group management defined for the CIO transport service has many similarities to the model we describe in this paper. A user of the CIO multi-peer transport service uses two different components for accessing the transport service and the transport group management service. Communications are handled with two completely separate

protocols. However, CIO transport group management is limited to one communications platform (ie depends on the usage of the CIO transport service), and it has not been implemented. Furthermore, since the X.500 directory service is proposed as a basis for the transport group management service, it will be impossible to have notifications sent to users, since X.500 is not capable of DSAs actively sending data to DUAs.

ITU's T.120 series of recommendations [13] is an example for a standardized architecture which also incorporates group and session management functionality. The basis of the T.120 infrastructure as shown in Figure 2 are the network specific transport protocols defined in T.123, which at the moment support data transfer using integrated services digital networks (ISDN), circuit switched digital networks (CSDN), public switched digital networks (PSDN), and public switched telephone networks (PSTN). Extensions to include future broadband networks are under study. T.123 is used by the multipoint communications service T.122/T.125, which defines a network independent service with flexible data transfer modes (broadcast and request/response), multipoint addressing (one to all, one to sub-group, and one to one), and multipoint routing (shortest paths to each receiver and uniform sequencing). Recommendation T.124 then defines a generic conference control which uses T.122's multipoint communications service. The abstract services of the conference control include create, query, join, invite, add, lock, unlock, disconnect, terminate, eject user, and transfer services. These services provide a powerful environment for implementing conferencing applications, which then use T.124 and T.122 services. However, the applicability of these recommendations is limited because only the transport infrastructures defined in T.123 can be used. The T.120 series of recommendations can therefore be regarded as one specific example which should be kept in mind when designing more general group and session management services.

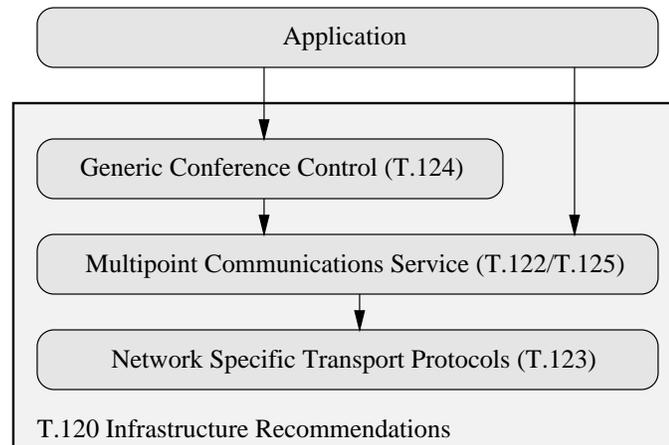


Fig. 2. Architecture of the ITU T.120 infrastructure recommendations

Other examples of multimedia communication systems dealing with group support have been described by Mauthe et al. [17]. However, all these approaches either do not concentrate on group and session management or they are restricted to certain transport infrastructures, or both. We therefore see the necessity to define a group and session management service which is independent from the transport infrastructure being used and provides a very abstract model of group communications.

3 Group and Session Management Requirements

Within this section, we identify the tasks related to group and session management which a group communication framework needs to perform. Group and session management in this context encompasses the storage and management of all information which is related to users (being the individuals working with applications), groups (which are sets of users and/or other groups), and sessions. Sessions are the representation of actual collaborations (instantiated by data exchange between distributedly working users) and can consist of one or more data flows (eg audio and video).

The main goal of the group and session management system (GMS) project is to design and develop a component which fills the gap between multipoint transport infrastructures as they are designed in actual research projects, and programmers of distributed multimedia applications. Another goal is to develop the infrastructure which is used by such a component. We therefore aim at providing these programmers with an API which is more powerful than the interfaces of common multipoint transport infrastructures. Our approach is to create a component which can be included into group communication frameworks in order to enhance their functionality. Figure 3 shows how this approach may be used, where GUA stands for GMS user agent and identifies the component provided by GMS. This component interacts with security and resource management components inside the group communication framework. It can be seen that the component is only one part of the group communication framework. Other components may be used for security purposes (in order to provide secure data communications), or for resource management, which could also include the mapping of application-level QoS parameters to network-level QoS parameters. However, we will now concentrate on the issue of group and session management.

What are the requirements such a component needs to fulfill? First and foremost, the data model used must be general enough to cover the majority of possible applications of the group communication framework. This means that both the objects and the available operations must be designed to allow a broad range of distributed multimedia applications. The abstractions needed must encompass users, groups of users, data connections between user groups, and a grouping mechanism for these data connections, because many applications need multiple data streams and it would be helpful to be able to handle these

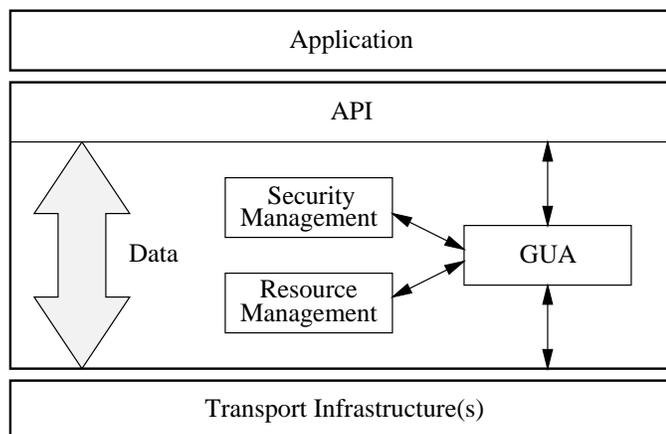


Fig. 3. Group and session management inside a group communication framework

together, especially with respect to authorization¹ and admission control² issues.

Furthermore, the component must be easy to integrate into existing group communication frameworks, and it must also be easy to exchange the transport infrastructure the component is using. This is a requirement because we do not want to restrict our model of group and session management to a single transport infrastructure. It has to be possible to incorporate the group and session management component into different group communication frameworks and to still be able to share as much information as possible. Naturally, transport specific information (such as addresses or QoS parameters) can not be used by group communication frameworks based on different transport infrastructures, but a lot of information, such as user identities, user authentication information, groups, and access rights, is useful independently of the transport infrastructure being used.

So far, we have mainly discussed the topics which are relevant for the support of collaborative (or distributed) applications. The aspect of multimedia communications requires additional support. However, it can be seen as a special case of communications, where data is being transmitted which requires handling with certain properties such as delay and delay jitter. In order to support distributed multimedia applications, it must be possible to specify the properties of a data connection in a way which is appropriate for multimedia data. The main idea in this area is the utilization of Quality-of-Service (QoS) parameters which make it possible to specify properties of a data connection. Because mul-

¹ Authorization control is performed to check whether a user is allowed to participate in a certain group communication. It is based on proper authentication of users and access privileges being assigned to objects.

² Admission control is used to check whether it is possible for a user to participate in a certain group communication. Admission control typically fails if there are not enough resources, such as local processing power or network capacity.

multimedia data connections may have interdependencies, it must also be possible to specify these. Examples for these interdependencies are synchronized data connections, which are common when individually transmitting audio and video data for video-phone applications. Another example would be the dependency of data connections when using hierarchical encodings.

As a conclusion, for the support of group and session management tasks of distributed multimedia applications, we need a system which provides connectivity throughout the lifetime of data connections, which may actively send notifications to applications, and an appropriate data model for the data inside the system. In the next section, we will discuss the model and an architecture for such a system.

4 A Model for Group and Session Management

From the requirements listed in the last section, several conclusions can be drawn. One is that a permanent database is required, which can store information about entities which are not permanently active in the context of a group communication framework, ie users and user groups. Another requirement is that certain events regarding a user of the group and session management system must be communicated to the user, and this must be initiated by the group and session management system. Therefore, it is appropriate to model the connection between the user and the group and session management system as a permanent connection during the lifetime of a user's work with a group communication framework. However, the data path of the user's application is entirely independent from the group and session management system, which distributes data with its own mechanisms. The resulting architecture of such a model is depicted in Figure 4, with GMS being the group and session management system.

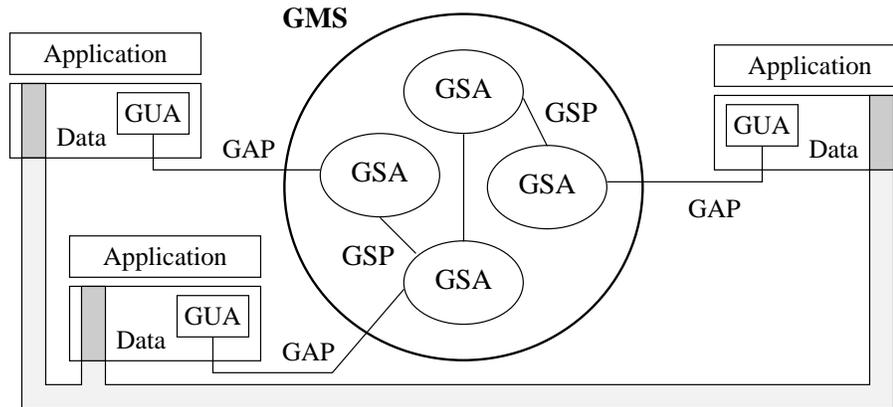


Fig. 4. GMS architecture

In this figure, GAP is the GMS access protocol, which is used as an entry point to GMS. GSP is the GMS system protocol which defines how data is exchanged inside the GMS. GAP is described in more detail in Section 4.1, GSP in Section 4.2. The main architectural components of GMS are GMS user agents (GUA) and GMS system agents (GSA), which will be described in Sections 5.1 and 5.2 respectively. While GUAs are included in group communication frameworks using GMS, GSAs are stand-alone components which, in their entirety, make up the GMS database. They are organized into domains, which are hierarchically ordered. The GMS architecture looks very similar to well-known directory services such as the Internet Domain Name System (DNS) [18] or ITU's X.500 directory service [14], but there are some notable differences.

DNS is based on the assumption that a simple lookup is sufficient (which is true for the purpose of DNS entries), while GMS requires a permanent connection between a user and the system because of the notifications which have to be delivered to GMS users. Furthermore, DNS is (for the normal user) read-only and anonymous, while GMS entries need to be modifiable and users normally are identified (and authenticated, if necessary). Therefore, the main difference between DNS and GMS is that DNS lookups are very short connections, while GMS GAP connections exist over the lifetime of a data connection, and that the typical GMS user has an identity which can be used for additional functionality, while a DNS user has anonymous access to DNS data.

The X.500 model is closer to the requirements of GMS than DNS. In fact, we first considered to use X.500 as a base for GMS, which then would have consisted simply of a set of X.500 object definitions and a software component granting access to these objects. However, there are some drawbacks with X.500, the two main disadvantages being the non-existence of DSA-initiated operations, which are needed for the notifications which are part of the GMS model, and the slow propagation of information inside X.500. Furthermore, X.500 requires the OSI stack of communication protocols, which does not fit our requirement to be able to use the access protocol over different transport infrastructures. We therefore decided not to use X.500 but to define a distributed directory service which exactly fits the needs of the GMS model. The main difference between X.500 and GMS is that GSAs are able to become active in GAP connections (instead of the purely reactive DSAs), and that GSP is designed for faster propagation of information than DSP. It is also easily possible to use GAP over a variety of transport protocols, provided they offer a reliable, connection-oriented service.

The most important definitions for GMS are the data types which are used inside the system. Because the design goal was to create a versatile model which can be used by a wide variety of distributed applications, the following object types have been defined. A complete definition of the object types can be found in the specification of GAP [23].

- *User*. A user is a person or entity using GMS. Each user has an identity (a name) and one or more ways of authenticating himself. This authentication may vary from no authentication at all (ie it is sufficient to use the right name) to sophisticated, hardware-oriented authentication schemes with mul-

multiple challenge iterations. A user object contains information about a user, such as his real world name, a description, his email address, and a list of the bindings of a user, ie the list of active GMS connections a user has.

- *Group*. GMS groups may consist of users and/or groups, depending on the definition of the group. Joining and leaving a group depends on the group's join policy and authentication requirements. Joins and leaves may be notified to a group's managers and/or members. Each group object may contain a group's real world name (eg the name of a company or a company's department), a description of the group, a group's mail address, and the access rights, which determine who is authorized to modify the attributes of the group.
- *Flow Template*. For several applications and communication platforms it is useful to have a number of predefined possibilities to set up connections. Flow templates contain information about data types which may be carried by a flow of that type, the necessary transport service, data which is needed to set up a flow of that type, information about uni- or bidirectional services, and a set of QoS parameters, which can be used to give a description of the flow template. However, flows may also be created without using a flow template.
- *Flow*. A flow is one connection for data transport. Depending on the flow's definition, it is either uni- or bidirectional, has a limited number of senders and/or receivers, and a renegotiation policy, which determines who is authorized to initiate QoS renegotiations for that flow. Flows are created when a session is created and are deleted when a session is deleted. Joining a flow takes place when a session is joined, and a flow is left when the session of a flow is left.
- *Session*. The main metaphor for group communication is a session. Each session is used to logically group a number of flows and to create an abstraction for management, authorization, and admission control for flows. The flows of a session are created when the session is created and deleted when the session is deleted. When joining a session, not all flows of the sessions must be joined, so users can choose which flows to use. Sessions may have application specific information, which consists of an application identification and application specific data, which may be interpreted by the application. Furthermore, the duration of a session may be given with either start or end times or both. In addition, it is possible to specify which authentication level a user must have to successfully join a session (provided he is authorized sufficiently). Authorization is based on the session's join policy which may be open (everyone may join), group (only members of the group associated with the session may join), or managed, which may be either relative (a given percentage of managers must confirm) or absolute (a given number of managers must confirm).
- *Certificate*. Applications with special security requirements may have the need to store certificates inside the GMS, which are used for checking data identity and integrity. Certificates include the type (which may be a prede-

finned type or any other type), the name type (which also has a number of predefined values and the possibility to define own types), the certificate's validity, a simple name, and data and signatures, which contain the informations which is necessary for checking the data.

In the context of distributed multimedia applications, another important issue is the one of Quality-of-Service (QoS) Parameters. GMS has a very general concept of QoS specification and usage, which ranges from no QoS (which would be used for TCP/IP) up to a arbitrary number of QoS parameters which can also be renegotiated. All participants of a flow are notified of a renegotiation, so that is possible for them to take appropriate actions, such as changing the protocol parameters to adapt to the new QoS values.

GMS has four QoS parameter types, which are *unsorted values*, *sorted values*, *integer values*, and *real values*. Unsorted Values are a set of predefined values, which are not in any particular order (an example for this is a QoS parameter which defines a coding algorithm, where it is not possible to arrange the different algorithms in any order). Sorted Values are also predefined values, but these values are given as a sequence, because it is possible to arrange them in an order (an example for this type of QoS parameter is the selection of a error detection algorithm, which may be given as a sequence of none, CRC8, CRC16, and some more sophisticated algorithms). Integer and real values represent the two basic types of numbers which may be used (which may be used for throughput or error probability numbers).

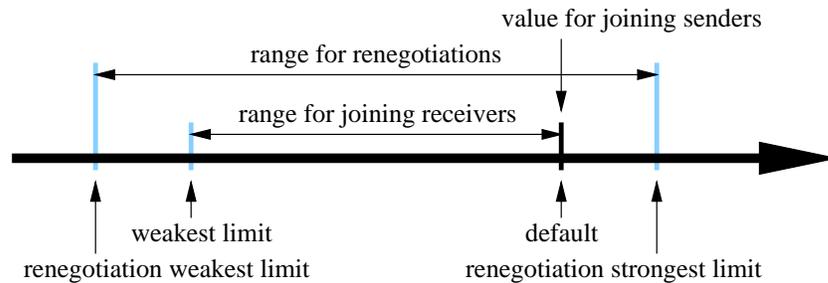


Fig. 5. Values and ranges for QoS parameters

Each flow's QoS parameters are defined by their name and type and at least a default value (which is the value used for joining participants if no local modifications are requested). Optionally, a weakest limit for joining the flow and strongest and weakest limits for renegotiations may be defined. For unordered values QoS parameters, there is no order of the values, therefore the join and renegotiation values must be given as sets of values. The concept of QoS parameter values and ranges is shown in Figure 5. When joining a session, the QoS parameters being used are taken from the flows' QoS definitions. According to the user's requirements, a weaker value than the default may be selected, as long

as it does not fall short of the weakest limit for joining the flow. However, this is only possible for receivers, senders must always join with the QoS value given as the default value.

In the following sections, we will describe the two protocols of GMS, namely GAP and GSP, as shown in Figure 4. These protocols, together with the object types described earlier in this section, form the main part of the GMS model.

4.1 GMS Access Protocol (GAP)

The GMS access protocol (GAP) is used by GMS user agents to connect to the GMS. While a large part of the protocol deals with operations initiated by the GUA, there are also some messages which are initiated by the GSA the GUA is connected to. Basically, GAP requires a reliable, connection-oriented transport infrastructure. Ideally, this transport infrastructure would also allow for secure communications, but is up to the GMS user to decide whether he requires a secure channel or not. However, the security mechanisms of GMS may be compromised if no secure GAP connection is being used. The prototype implementation of GAP is based on TCP/IP. However, this requires the group communication framework (the GUA is part of) to use TCP/IP, which could be a limitation.

If a new transport protocol for GAP should be used, both the GUA's network adaptation layer (as described in Section 5.1) and the GSA's GAP server component (as described in Section 5.2) had to be extended to support the new protocol. This way, one could easily think of a multi-protocol GSA, which for example would allow TCP/IP based as well as ISO/TP4 based GUAs to connect to it. The only limitation is that a reliable, connection-oriented protocol is required. However, it should always be kept in mind that data connections and GMS (ie GAP) connections are completely independent (as shown in Figure 4), so it is not always necessary to adapt GAP to a new transport protocol.

GAP itself can be split into three phases, each of them dealing with different aspects of the GUA-GSA interaction. A detailed specification of GAP can be found in [23]. The three phases of GAP are as follows.

- *GUA binding phase.* This is the initial phase of GAP, which is entered directly after a GUA has connected to a GSA. In this phase, the GUA has to bind to the GMS, ie it has to register with the GSA. It sends GAP version information and gets as a reply the maximum number of users supported by the GSA³. Furthermore, the domain name of the GSA is also sent to the GUA.
- *User authentication phase.* After the GUA has bound to the GMS, it enters the GUA bound state and is now ready to bind users. User authentication in GAP ranges from none to complex authentication schemes which require

³ It is possible that such a maximum does not exist when binding to the GMS (eg because the maximum is evaluated dynamically), in this case no such number is given.

multiple data exchanged between the GSA and the GUA (ie the user)⁴. One common way of authentication is the Unix password scheme which authenticates a user by simply checking a password. However, as a result of the user authentication phase, the user is either successfully authenticated and thus bound to the GMS, or the binding attempt failed.

- *User bound phase.* Only a user who previously bound to the GMS can use the majority⁵ of GAP commands. These are commands to directly access the GMS objects (such as create, modify, or delete an object), or commands which implicitly modify objects and relations, such as join and leave for groups and sessions. The authorization to perform these commands is determined by certain policies defined within the objects and the identity of the user⁶. Furthermore, in this phase it is possible that notifications about certain events are sent to the GUA.

It is important to notice that there are two possibilities to bind multiple users to the GMS. The first possibility is that every user uses his own GUA, which then opens a GAP connection to the GSA. However, this approach may fail if only one group communication framework is running on a machine which is used by several users. In this case, it is possible that several users bind to the GMS using the same GUA. GAP has been designed in a way that multiple users do not interfere when using the same GAP connection. However, because of this design, it is possible that one user is in the user authentication phase while another user is in the user bound phase. For this reason, GAP is specified using parallel state machines. A more detailed description of this design can be found in the GAP specification [23].

4.2 GMS System Protocol (GSP)

The GMS system protocol (GSP) is used by GMS system agents to communicate. This communication is necessary to exchange data and to exchange information about the configuration of GMS. A detailed description of GSP can be found in the GSP specification [24].

One of the main points about GSP is that it is a multicast-based protocol. We use the reliable, FIFO ordered (according to the definitions given by Hadzilacos and Toueg [8]) multicast protocol described by Bauer and Stiller [2] as base for GSP. The usage of a multicast based protocol is very efficient because GSAs are

⁴ One example for a class of authentication schemes which require this type of data transfer are challenge-response schemes, which depend on a sequence of challenges sent by the server to which the client has to respond. Only if all challenges are answered correctly, the authentication is successful.

⁵ Exceptions are the commands of the two other phases, which form a small subset of GAP commands.

⁶ Because users may be bound to the GMS using more or less strong authentication mechanisms, GMS includes the concept of *authentication requirements*, which define for each object an authentication level by which a user must at least be bound to perform any operation regarding this object

grouped into hierarchically ordered domains, and all GSAs of a domain can be reached with a single multicast address. This way, we can use true multicasting as opposed to the multicasting mode of the X.500 directory as described in X.518 [15]. We use multicast as the method for distributing requests to all GSAs of a domain. However, when replying to a request, the replying GSA uses unicast, thus only sending the reply to the originator of the request. This approach minimizes the network load caused by the interacting GSAs.

One concept used for GSP is the one of tokens. Tokens exist inside a domain and are used to determine which GSA inside a domain is authorized to perform certain operations. The three token types of GSP are as follows, where each token exists for every domain inside the GMS.

- *Propagation of requests.* Because it is often necessary to propagate domain name resolution requests (which are described in detail later in this section) either up or down the domain hierarchy, there has to be one GSA inside each domain which is responsible for this task. Whether this role is fixed or moved from one GSA of a domain to another using some kind of load balancing strategy, is outside the scope of the GSP specification.
- *Object creation.* Object creation is also handled by multicast requests. Because only one GSA is allowed to actually create a new object when requested (otherwise duplicates would be created), the task of object creation also depends on a token. This token may be rotated among a domain's GSAs using a strategy which takes into account the storage space available on each GSA.
- *Forwarding and processing of queries.* Queries are the most processing intensive operations inside the GMS because it is necessary to search for objects matching a given pattern. Queries must either be forwarded or processed inside a domain by a dedicated GSA, which collects the results and sends them back to the originator of the query. This role is also represented by a token and may also be assigned dynamically according to some strategy.

Because tokens may get lost (eg when the machine of the token holder crashes) or may be duplicated (eg if the network has been temporarily partitioned), it is always possible for a GSA to request a token renegotiation for a domain. For this task, each GSA implements a simple finite state machine which has the three states *monitoring*, *competing*, and *tokenholding*. *Monitoring* and *tokenholding* are two stable states, indicating a GSA which does not have a token respectively does have a token. When a token renegotiation request is sent to the domain, all GSAs enter the *competing* state. By replying with claim token messages, all GSAs try to get the token. Based on the content of the claim token messages, each GSA can decide which GSA will become the token holder. This GSA then enters the *tokenholding* state, while all other GSAs will become *monitoring*. The token renegotiation process is initiated by a GSA requesting a service from a domain's GSAs and either getting no reply (ie the token was lost) or more than one reply (ie the token was duplicated) after a predefined period of time.

The basic process of operations being carried out within GSP can be separated into two phases. The first phase is the domain name resolution. As men-

tioned before, domains are hierarchically ordered. The GSAs of each domain only know the address of their directly superior domain and the addresses of all directly inferior domains. There is no such thing as the top-level domain but a set of top-level domains, where each top-level domain knows the addresses of all other top-level domains and the addresses of all directly inferior domains. Whenever a GSA wants to send a request to another domain, it first checks whether it has cached the address after a previous request. If not, the domain name resolution phase is started. Depending on whether the required domain is hierarchically above the requesting GSA or not (which can be decided based on the domain's name), the domain name resolution request is either sent to the superior domain or to the appropriate inferior domain. In this domain, the GSA holding the *propagation of requests* token will forward the domain name resolution request to the next domain and reply with a domain name resolution pending message to the requesting GSA. If this pending message (or more than one) is not received after a certain timeout, the requesting GSA will send a token init request to the domain, initiating a token renegotiation for this domain (as discussed in the previous paragraph). This process, which is shown in Figure 6, continues until the address of the requested domain is found, which is then directly sent back to the initiating GSA.

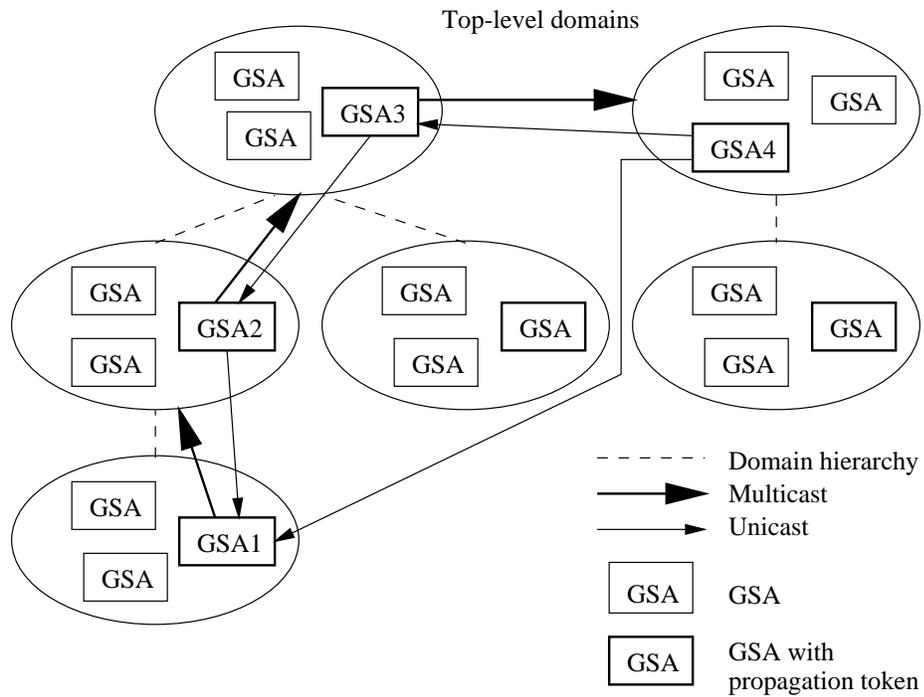


Fig. 6. GSP domain name resolution

In this figure, GSA1 is the GSA initiating the domain name resolution. GSA2, GSA3, and GSA4 each reply with a domain name resolution pending message to the GSA which sent the request to the domain using its multicast address. GSA4, which finally knows the address of the domain to which GSA1 wants to send a request, directly responds to GSA1 with the domain's address. If such a reply is not received after a timeout, the domain name resolution process is initiated again. Both this timeout and the timeout which is used to send the token init request must be chosen carefully to find the optimal balance between unnecessary repetitions respectively token renegotiations and too long idle periods. However, since we assume that the domain hierarchy will be relatively flat (eg as deep as the DNS domain name space, with a similar organization), there are not too many GSAs involved in the domain name resolution process.

Once a GSA has resolved the address to which a request must be sent (either by the process described above or from a cache, which contains addresses accessed before), the operation itself can be carried out. This is done by sending the request containing the operation to the domain. Depending on the operation, the GSAs of the requested domain behave differently. For example, when sending a modify request, the GSA storing the object will be the only one responding to the request, while all other GSAs of the domain silently ignore the request. This is possible because it is clear that only the GSA storing the object in its local database can process the request. On the other hand, if a create request is processed, first an object present request is sent to all GSAs of the domain. All GSAs of the domain send a reply, indicating whether they hold the *object creation* token or not⁷. This is necessary because the requesting GSA must be able to detect whether there is exactly one token holder. If all GSAs⁸ reply with an indication that they do not hold the token (or if more than one token holder is answering), it can be concluded that a new token holder must be found and the GSA sends a token renegotiation request to the domain. Otherwise, the create request is directly sent (using unicast) to the GSA which indicated that it has the *object creation* token.

Additional protocol mechanisms exist for adding GSAs to and removing GSAs from a domain. Normally, it is assumed that a GSA is a permanently running process, but even then occasional interrupts (eg if the machine a GSA runs on is stopped) can occur. If a GSA is started, it must first join the multicast group which is assigned to the domain it is joining. Then the GSA has to send a join domain request to this address, and all GSAs of the domain reply to this request. The first reply to that request, sent by any GSA inside the domain, concludes the start up procedure and the GSA becomes part of the domain. The remaining join domain replies can be ignored. Each GSA has a local table of

⁷ This procedure is feasible under the assumption that the number of GSAs per domain is moderate. Again, we consider an architecture similar to that of DNS, where typically each domain is served by very few servers. If the number of GSAs would be much greater than 10, the approach taken would have been prohibitive.

⁸ The fact that all GSAs have replied can be concluded from the total number of GSAs in the domain, which is contained in every reply sent by a GSA.

which contain a list of triples containing an unique GSA identification, a flag indicating whether it is currently active, and a version number. This table is updated if a join domain request is received. This table is sent periodically to the domain, so that inconsistencies finally disappear. If a GSA wants to leave a domain, it sends a leave domain request to the domain, waits for the first leave domain reply, and then leaves the multicast group. The remaining GSAs update their internal tables according to the leave domain request. These tables are also updated if differences are detected between the periodically received version and the local version.

5 Implementation

In the previous section, we described the model and an architecture for the group and session management system GMS. In this section, we will briefly describe the implementation of our GMS prototype. At the time of writing, the GUA implementation has finished and the GSA implementation is still under way. As implementation base we use Sun workstations running Solaris 2.5, the Sun version of Unix. As tools we used StateMate, a commercial software for designing, simulating, and generating code for finite state machines, and snacc by Sample and Neufeld [20, 21], a public domain software for generating code for ASN.1 coding and decoding. ASN.1 coding and decoding software was required because we use ASN.1 [12] as notation language for the syntax in our protocol specifications.

5.1 GUA

The implementation of the GMS user agent, which is, together with a more detailed description of the concepts, described in [25], focuses on two aspects, the first being the easy adaptability of the GUA code to another transport infrastructure. We therefore used only abstract procedures for accessing the transport infrastructure and created a network adaptation layer which can easily be modified as long as the new transport infrastructure being used provides reliable, connection-oriented communications. This design can be seen in Figure 7.

The other important aspect is the one of control flows. Because normally the GUA can get input from both the user (at the programming interface layer) and the network (at the network adaptation layer), and we did not want to delay processing of one direction until the other direction has been handled, we use a multi-threaded design, where one thread is responsible for processing requests from the programming interface layer, while the other thread processes requests coming from the network adaptation layer. The reason why we decided to use this solution is because, as mentioned in Section 4.1, it is possible that multiple users use one GUA, and it is important to make sure that no user is able to disturb other users' work with the GMS.

The implementation of the GUA component consists of seven main building blocks. Network adaptation and programming interface are cleanly separated

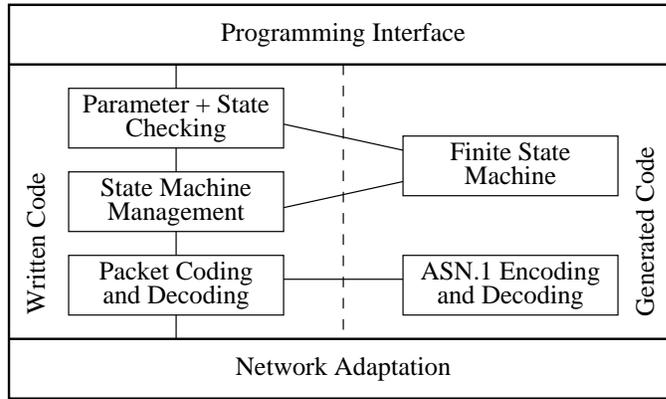


Fig. 7. GMS user agent (GUA) design

to make the code easily adaptable to different environments. Furthermore, we use generated code for the GUAs finite state machines and the encoding and decoding of GAP PDUs. This code is controlled by three components, which are responsible for checking parameters and the state of the GUA's state machine, the state machine management (ie performing state transitions and executing the appropriate actions), and the marshaling and unmarshaling of arguments.

5.2 GSA

The GSA implementation consists of three main building blocks, which are the implementation of the two protocols GAP and GSP, and the internal logic, which is responsible for performing the actions which are requested by either a GUA or a GSA. It is also possible to have GSAs which do not provide GAP access. In this case, the building block containing the GAP functionality is omitted. This architecture is depicted in Figure 8, which goes into a little more detail. In this figure, each labeled block represents a process, while the MQ components are Unix message queues, providing a convenient interprocess communication mechanism.

The main component of the GSA is the GSA manager which is the process started first during initialization. If the GSA supports GAP, the GSA manager starts the GAP server, which is then able to accept connect requests from GUAs. The GAP server currently uses TCP/IP, but it could be easily extended to support other transport protocols. The GSA manager also starts multicast processes for sending to and receiving from multicast groups. These processes implement the reliable multicast protocol mentioned in Section 4.2. Furthermore, the GSA manager starts a unicast receiver process which handles all incoming requests for the GSA. These are replies to multicast requests. The multicast modules are implemented on top of UDP/IP multicast, while the unicast receiver is based on

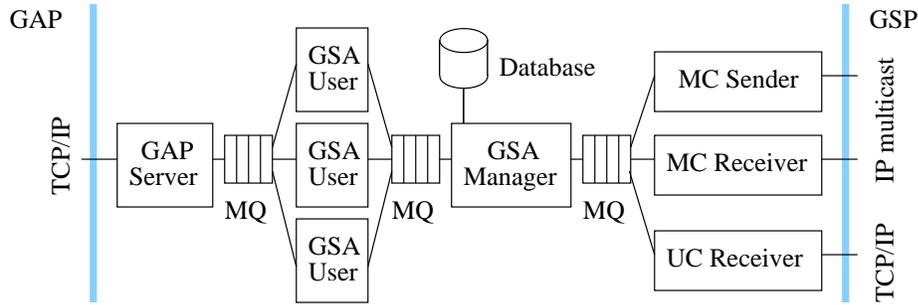


Fig. 8. GMS system agent (GSA) design

TCP/IP⁹. Finally, the GSA manager is the process which has direct access to the local database.

The GAP server handles the *GUA binding phase* described in Section 4.1. If a GUA initiates the *user authentication phase*, the GAP server dynamically starts a GSA user process which is then responsible for handling this users authentication and, if the authentication has been successful, also the *user bound phase*. The GSA user process terminates if the user unbinds, if the user's authentication failed or if the GUA forces an unbind (ie the GAP connection is terminating). Consequently, for each user bound to the GSA, a GSA user process exists which handles the users requests and delivers any results or notifications to the user.

The GSA manager has the role of the central entity in the GSA design. It accepts all incoming GSP PDUs and processes them according to their content. If a PDU is the result of a operation requested by a GSA user process, it is forwarded to this process. If an incoming GSP PDU requires the GSA to perform some actions, it either performs the appropriate actions itself or it starts a separate process (not shown in the figure) which lives as long as the operation is being processed. The GSA manager also has direct access to the database, where all local objects and relations are stored. Currently we use the standard Unix ndbm package¹⁰ for managing the database. This package implements a simple storage for key/data pairs, in our case the pairs consist of an object's name as the key and the object's ASN.1 representation as data.

⁹ Because we use the unicast connection only for sending a reply to the requesting GSA, a transactional variant of TCP such as T/TCP described by Braden [4] would be preferable. However, at the moment we use standard TCP, thus including the overhead of the rather expensive TCP connection establishment and the problem with both ends going to the TIME-WAIT state after closing the connection.

¹⁰ The ndbm format is the new format for Unix databases which replaces the older dbm format and library.

5.3 Implementation results

The implementation described in the previous sections has been tested in various ways. The majority of test has been performed as load testing, where a huge load of requests was produced and the behavior of the system has been observed. After these tests (which included a number of test domains with GSAs in the local network and internationally distributed), all timeouts have been adjusted carefully to find the optimal balance between unnecessary repetitions and unnecessary wait periods. Most operations now have five or ten seconds timeouts, assuming that due to the usage of a reliable multicast protocol (which already has internal timers for keep-alive packets), timeouts on GSP level should occur only rarely.

Analyses of the GSA code showed that about 70% of the time spent is used for program logic, 30% is used for database accesses, and only 0.3% are used for coding and decoding ASN.1 data. This was a surprise to us, since the code generated by the Snacc ASN.1 to C/C++ compiler is huge (120000 lines of generated C++ code as opposed to 20000 written lines of C++ code for the various GSA processes). However, this still causes problems, because although the CPU load caused by a GSA running on a system is moderate, due to the size of the processes (most of them including coding/decoding routines), a system running a GSA is heavily loaded by swapping processes from and to memory.

One important point when discussing the performance of a distributed directory service is the scalability of the architecture. GMS can be scaled in three dimensions, which are discussed in the following list.

- *Number of users per GSA.* The current GSA implementation is obviously not suited to support a larger number of users, since every user is represented by a process (as shown in figure 8). However, the resources required for each user could be reduced to a few table entries if the GSA code was designed appropriately. Thus, the number of users per GSA could be fairly big (in the magnitude of a few hundreds) if the GSA implementation was carefully designed.
- *Number of GSAs per domain.* The number of GSAs per domain also is the number of GSAs receiving all multicast requests to this domain and replying to them, if necessary. Therefore, the number of GSAs per domain should be kept fairly small (in the magnitude of ten) to avoid the well-known implosion problem. This imposes no problem, since GSAs are meant to be central services which are remotely accessed using the GMS access protocol (GAP).
- *Number of domains.* The number of domains can be scaled in two ways. Extending the domain hierarchy horizontally does not cause any change in GSP performance, since domain name resolution is not affected by the number of subdomains of a domain. Extending the domain hierarchy vertically (ie introducing new levels of subdomains) influences the domain name resolution linearly, since the domain name resolution requests have to be propagated through more domains. Requests to domains are not influenced at all, since they are directly addressed to the domain. Furthermore, when using caching

in the GSA instead of performing a domain name resolution for every request, the effect of extending the domain hierarchy vertically could be minimized. Hence, the number of domains does not influence GMS in a way which could cause performance problems.

Consequently, GMS is able to be used in a large scale, provided the number of GSAs per domain is kept reasonably small (which is also preferable from a management point of view). We therefore believe that the approach to group and session management presented in this paper not only makes group communication platforms more flexible, but also can be used in a global scale.

6 Conclusions

In this paper we describe a group and session management system (GMS) for distributed multimedia applications. The GMS model assumes that a special component, a GMS user agent (GUA), is included into group communication frameworks which want to incorporate GMS functionality. This component then becomes an integral part of the group communication framework, ie it is not possible for application programmers to access the GUA directly. This approach has been chosen because a number of operations (especially the join session operation, which joins the requester to a number of data flows) can not be completely processed inside the GUA, but also need the group communication framework (eg for performing an admission control which needs to check whether the local and network resources are sufficient to join a session). The exchange of user data and of GMS access protocol (GAP) data is performed independently, so it is possible to use different transport infrastructures for data exchange and GAP connectivity.

Furthermore, the GMS data model is designed in a way that it allows the modeling of users, groups, and sessions in an abstract way which is suitable for different group communication frameworks. It is therefore possible to easily integrate GMS functionality into group communication frameworks, the two main issues being the abstract data model of GMS, and the separation of data and control information (GAP). The abstract data model can be used to map a framework's internal model of connections and connection handling to the abstractions used by GMS, which is then accessible to all GMS users, even if they are using different group communication frameworks. The separation of data and control information provides the framework designers with the opportunity to separate data exchange and management information, which can even be transferred using different transport infrastructures.

In addition to the standard GMS usage, where a GUA is integrated into a group communication framework, we are also implementing an application which only uses the GUA for communications and consequently can not be used for data transfer. This application could serve the same purposes than the Internet's mbone session directory (as implemented by the sd/sdr tools), but with a richer functionality (such as authentication, authorization, and the

ability to use more than one group communications framework). At the time of writing, this application is in the implementation phase.

The GMS architecture is that of a distributed directory service. The distributed components are GMS system agents, communicating via the GMS system protocol (GSP). GSAs are grouped into hierarchically organized domains, which reflect organizational structures in the real world. GSP is a multicast protocol, with the unit of addressing being the domain. The protocol design is based on the assumption that domain are relatively small with respect to the number of GSAs (not much more than 10 GSAs in one domain), and that the hierarchy is relatively flat (not much more than 5 levels). Based on observations of the current structure of DNS and X.500, we believe that these assumptions are realistic. We will start an experimental GMS as soon as the GSA implementation is finished, which will give us the opportunity to adjust the timeout values (which are crucial for the proper operation of GSP) and to evaluate an internationally distributed version of GMS.

The authors would like to thank Daniel Bauer and Gerhard Nigg for providing us with the software for the reliable multicast protocol GSP is based on. We also would like to thank Murali Nanduri who implemented most of the GUA software.

References

1. Michael Altenhofen, Jürgen Dittrich, Rainer Hammerschmidt, Thomas Käppner, Carsten Kruschel, Ansgar Kückes, and Thomas Steinig. The BERKOM Multimedia Collaboration Service. In *Proceedings of ACM Multimedia 93*, pages 457–463, Anaheim, California, 1993. ACM Press.
2. Daniel Bauer and Burkhard Stiller. An Error-Control Scheme for a Multicast Protocol Based on Round-Trip Time Calculations. In *Proceedings of the 21st Conference on Local Computer Networks*, Minneapolis, October 1996.
3. Daniel Bauer, Erik Wilde, and Bernhard Plattner. Design Considerations for a Multicast Communication Framework. In *Proceedings of the Tenth Annual Workshop on Computer Communications*, Eastsound, Washington, September 1995.
4. R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. Internet RFC 1644, July 1994.
5. C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware – Some Issues and Experiences. *Communications of the ACM*, 34(1):38–58, 1991.
6. Hans Eriksson. MBone: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, 1994.
7. Thomas Gutekunst, Thomas Schmidt, Günter Schulze, Jean Schweitzer, and Michael Weber. A Distributed Multimedia Joint Viewing and Tele-Operation Service for Heterogeneous Workstation Environments. In Wolfgang Effelsberg and Kurt Rothermel, editors, *GI/ITG Arbeitstreffen Verteilte Multimedia-Systeme*, number 5 in Praxis, Information und Kommunikation, pages 145–159, Stuttgart, Germany, February 1993. K. G. Saur.
8. Vassos Hadzilacos and Sam Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, New York, second edition, 1993.
9. M. Handley, J. Crowcroft, and C. Bormann. The Internet Multimedia Conferencing Architecture. Internet Draft, MMUSIC Working Group, February 1996.

10. Mark Handley and Van Jacobson. SDP: Session Description Protocol. Internet Draft, MMUSIC Working Group, November 1995.
11. Lutz Henckel. Multipeer Connection-mode Transport Service Definition based on the Group Communication Framework. Technical report, GMD FOKUS, Berlin, June 1994.
12. International Organization for Standardization. Information processing systems – Open Systems Interconnection (OSI) – Specification of Abstract Syntax Notation One (ASN.1). ISO/IS 8824, 1990.
13. International Telecommunication Union. Data Protocols for Multimedia Conferencing. Draft Recommendation T.120, 1995.
14. International Telecommunication Union. The Directory – Overview of Concepts, Models and Services. Recommendation X.500, March 1995.
15. International Telecommunication Union. The Directory – Procedures for distributed operations. Recommendation X.518, March 1995.
16. T. Kirsche, R. Lenz, H. Lührsen, K. Meyer-Wegener, H. Wedekind, M. Bever, U. Schäffer, and C. Schottmüller. Communication support for cooperative work. *Computer Communications*, 16(9):594–602, 1993.
17. Andreas Mauthe, Geoff Coulson, David Hutchison, and Silvester Namuye. Group Support in Multimedia Communications Systems. In D. Hutchison, H. Christiansen, G. Coulson, and A. Danthine, editors, *Teleservices and Multimedia Communications – Proceedings of the Second COST 237 Workshop*, volume 1052 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, November 1995. Springer-Verlag.
18. P. Mockapetris. Domain Names – Concepts and Facilities. Internet RFC 1034, November 1987.
19. T. Rodden, J. A. Mariani, and G. Blair. Supporting Cooperative Applications. *Computer Supported Cooperative Work*, 1(1–2):41–67, 1992.
20. Michael Sample. Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler. Technical report, University of British Columbia, Vancouver, July 1993.
21. Michael Sample and Gerald Neufeld. Implementing Efficient Encoders and Decoders For Network Data Representations. In *Proceedings of the IEEE INFOCOM '93 Conference on Computer Communications*, pages 1144–1153, San Francisco, 1993. IEEE Computer Society Press.
22. Henning Schulzrinne. Simple Conference Invitation Protocol. Internet Draft, MMUSIC Working Group, February 1996.
23. Erik Wilde. Specification of GMS Access Protocol (GAP) Version 1.0. Technical Report TIK-Report No. 15, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, March 1996.
24. Erik Wilde. Specification of GMS System Protocol (GSP) Version 1.0. Technical Report TIK-Report No. 19, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, September 1996.
25. Erik Wilde, Murali Nanduri, and Bernhard Plattner. A Transport-Independent Component for a Group and Session Management Service in Group Communications Platforms. In P. Delogne, D. Hutchison, B. Macq, and J.-J. Quisquater, editors, *Proceedings of the European Conference on Multimedia Applications, Services and Techniques*, pages 409–425, Louvain-la-Neuve, Belgium, May 1996.
26. Neil Williams and Gordon S. Blair. Distributed multimedia applications: A review. *Computer Communications*, 17(2):119–132, 1994.