# Specification of GMS System Protocol (GSP) Version 1.0

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

**Abstract**

Group communications require special support for name and address management, QoS support, and connection establishment. The group and session management system (GMS) is a distributed directory system which is specifically designed to support group communication infrastructures. This report briefly introduces the concepts of GMS, the data types available, and then gives the specification of GSP, the GMS system protocol. This protocol is used for communication between GSAs, ie it is used for GMS's internal communication. GSP is specified by state diagrams, time sequence diagrams, textual descriptions, the PDU syntax in ASN.1, and the PDU semantics in comments given for each PDU.

## Contents

# 1 Introduction

This report contains the specification of GSP, the GMS system protocol, version 1.0, which is used for communication between GMS system agents (GSA) within the Group and Session Management System (GMS). The general architecture of GMS is shown in figure 1. The GMS service offered by GMS system agents (GSA) to GMS user agents (GUA) is implemented by a number of GSAs communicating using the GMS system protocol (GSP). Users of the GMS (which will typically be programmers of communication infrastructures) will be represented by GUAs and may access any GSA using GAP. The specification of GAP is available in another technical report [21]. For more information about GMS and its goals it is possible to read some conference papers [2, 22, 23]. The major benefits of the GMS architecture can be summarized as follows.

- *Reduced implementation costs.* Because of the transport-independency of the GUA component, it could be used in different transport platforms without having to implement the functionality for each platform. This leads to a reduction of implementation costs for new platforms using this component.
- *Transport-independent naming.* The name to address mapping is one of the tasks of the what is often called the management plane of communication infrastructures (eg the QoS-Architecture of Lancaster University [5]). If naming is implemented by a transport-independent component, then it is possible to use the same names for addressing for different transport platforms. This would eliminate the situation of today, where each collaborative application has its own name space so that the use of more than one collaborative application can become a very complicated process in terms of user and group management.
- *Session directory functionality.* A session directory similar to the well-known mbone session directory would be possible, and it would be a more general directory. It would not only list the sessions and the respective applications, but also the transport infrastructure being used and users and groups. This way it would be easy to implement collaborative applications which are able to use different transport infrastructures.

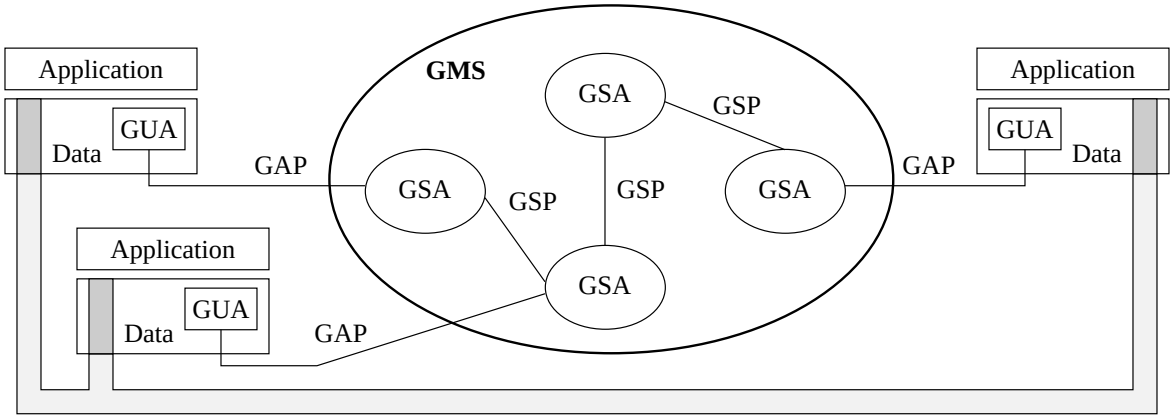


Figure 1: GMS architecture

To understand GSP, it is necessary to first introduce the object types used within the GMS. This is done in section 2. Also explained in this section are the QoS concepts of GMS, and the relations which may exist between GMS objects. The object types of GMS (described in section 2) as well as the PDUs which are exchanged when using GSP (described in section 4) are specified in ASN.1 as defined by the ITU [7, 10, 18]. Because ASN.1 only describes the syntax, the specifications

are accompanied by explanations of the semantics. Furthermore, section 3 contains descriptions of all GSP procedures, ie of the rules according to which communicating GSAs exchange GSP PDUs. The "`--snacc isPdu:"TRUE" --`" comments are necessary for the snacc ASN.1 compiler which is described by Sample and Neufeld [16, 17]. These comments direct the compiler to generate encoding and decoding routines for the marked data types. We use this compiler to check the ASN.1 definitions and to generate code for coding and decoding ASN.1.

Section 2 of this technical report is almost identical with the respective section of the GAP specification [21], because the GMS object types and relations are used within both protocols. Only some references to GAP specific sections and some typographic errors have been removed. Most of the protocol itself is the result of two diploma theses, written by Pascal Freiburghaus [6] and Daniel Koller [13].

The typographic conventions in this specification are very simple. `Text in typewriter type` refers to attribute types or values taken from ASN.1 definitions (either object type specifications or PDU definitions). Text in sans serif type refers to definitions from state diagrams, which may be states, events, or actions.

# 2 GMS Data

This section contains all definitions of object types and relations which are necessary to interpret GSP PDUs. The ASN.1 definitions are separated into four different modules. The first module (`GMS-COMMON` described in section 2.1) contains common definitions of types which are used in other modules. The second module (`GMS-QOS`) described in section 2.2 defines the QoS attributes available for flows. This section also gives a general overview over the concept of QoS used in GMS. Module `GMS-OBJECT-TYPES` (described in section 2.3) then defines the actual object types used inside the GMS. It uses definitions from `GMS-COMMON` and `GMS-QOS`. The last module described in this section (`GMS-RELATIONS` described in section 2.4) imports definitions from `GMS-COMMON` and describes relations which may exist between GMS objects.

## 2.1 Common definitions

The following ASN.1 module `GMS-COMMON` contains definitions of common data structures which are used for object type definitions (described in section 2.3), relation definitions (described in section 2.4), and definitions of GSP PDUs (described in section 4).

```
GMS-COMMON DEFINITIONS ::= BEGIN

AccessRight ::= SET {
    read                [0]     BOOLEAN,
    modify              [1]     BOOLEAN,
    delete              [2]     BOOLEAN }
```

The `AccessRight` data type is used for determining the rights that a user of a particular category has to manipulate an object. It is used in the definitions of the object type `User` (section 2.3.1), `Group` (section 2.3.2), and `Session` (section 2.3.5). The `read` access right allows a user to read attributes of an object. The `modify` access rights allows a user to modify an object. The `delete` access right allows a user to delete an object from the GMS.

```
Application ::= CHOICE {
    sessionDirectory    [0]     NULL,
    other               [1]     IA5String }
```

The`Application` data type is used to describe which application a GMS user is using (explained in the `User` object type described in section 2.3.1), or which application is used within a session (explained in the `Session` object type described in section 2.3.5). An application is either the session directory (an application which is only used for browsing through GMS data) or any other application described by a string.

```
AuthLevel ::= INTEGER

AuthRequirements ::= AuthLevel

AuthType ::= CHOICE {
    none                [0]     NULL,
    name                [1]     NULL,
    unixPassword        [2]     NULL,
    oneTimePassword     [3]     NULL,
    tokenChallenge      [4]     NULL,
    rsaChallenge        [5]     NULL,
    other               [6]     SET {
        authLevel           [0]     AuthLevel,
        authName            [1]     IA5String } }
```

Authentication is possible in different ways when binding to the GMS. Depending on the level of security an authentication method provides, the methods are being assigned `AuthLevel` values (ranging from 0 to 100). Because a number of well know methods are predefined, they are not assigned an explicit level. These methods are `none` (no authentication at all, the user is bound anonymously, level 0), `name` (the name must be known, but there is no authentication mechanism, level 10), `unixPassword` (a password must be submitted which is encrypted according to standard Unix password encryption, level 20), `oneTimePassword` (a password must be submitted which is only valid for one authentication, level 30), `tokenChallenge` (level 40), and `rsaChallenge` (level 50). Other methods may be used by using the `AuthType` value `other` and defining an explicit `AuthLevel` and a name as a string.

The `AuthRequirements` are used to determine which authentication level must have been used to accept the authorization of a user to access an object. The `AuthRequirements` are used for the object types `User` (described in section 2.3.1), `Group` (described in section 2.3.2), and `Session` (described in section 2.3.5). This combination of authorization and authentication makes it possible to create objects (groups and sessions, in particular) with relaxed security conditions and to also create objects which require a strong authentication method to be used by anyone who wants to perform any operation on these objects.

```
CertificateType ::= CHOICE {
    pgp                 [0]     NULL,
    x509                [1]     NULL,
    nis                 [2]     NULL,
    other               [3]     IA5String }
```

The `CertificateType` data type is used for the object type `Certificate` (section 2.3.6). It determines which type of certification is used with a certificate. Three predefined values may be used (`pgp`, `x509`, and `nis`), if the certificate uses another kind of certification, `other` may be used and the type is described by a name given as a string.

```
DataType ::= CHOICE {
    audio               [0]     NULL,
    video               [1]     NULL,
    data                [2]     NULL,
    other               [3]     IA5String }
```

The `DataType` of a `FlowTemplate` (section 2.3.3) is used to characterize the data which may be transported with a flow of this flow type. Three predefined values may be used, defining `audio` data, `video` data, or raw `data`. If another data type needs to be specified in a `FlowTemplate`, the `other` type may be used and the data type is described by a name given as a string.

```
GmsDomainName ::= SEQUENCE OF IA5String

GmsObjectName ::= SEQUENCE {
    relativeName        [0]     IA5String,
    gmsDomainName       [1]     GmsDomainName }

GmsRelationName ::= SEQUENCE {
    relativeName        [0]     IA5String,
    gmsDomainName       [1]     GmsDomainName }
```

Names inside the GMS are hierarchically organized (like the name space of the DNS [14] or the concept of distinguished names of the ITU's X.500 directory [12]). A `GmsDomainName` simply is a sequence of strings, where the first string is the innermost domain and the last string is the outermost domain. Objects and relations of the GMS have names which are unique inside one domain. So each `GmsObjectName` and each `GmsRelationName` is a sequence of the object's name (which is similar to an X.500 RDN) and a `GmsDomainName`, which determines in which domain this object is located.

```
GmsObjectInformation ::= IA5String
```

Each object inside the GMS has `GmsObjectInformation` as one attribute. This attribute can be used to give a short description of the purpose or the meaning of an object. The description is simply a string of characters.

```
GsaAddress ::= OCTET STRING

GuaRelativeAddress ::= OCTET STRING

GuaAddress ::= SET {
    gsaAddress          [0]     GsaAddress,
    guaRelativeAddress  [1]     GuaRelativeAddress }
```

GUAs and GSAs have addresses which are used for finding the respective components. A `GsaAddress` simply is an octet string and gives the address of a GSA in a way specific to the GSP. A `GuaAddress` simply is a set of a `GsaAddress` and a `GuaRelativeAddress`, which is sufficient for addressing a GUA because every GUA bound to the GMS does so by binding to exactly one GSA. The `GuaAddress` is used in the definition of the object type `User` (described in section 2.3.1).

```
NameType ::= CHOICE {
    rfc822              [0]     NULL,
    e164                [1]     NULL,
    ipv4                [2]     NULL,
    other               [3]     IA5String }
```

The `NameType` data type is used for certificates, where it determines of which type a name of a certificate is. There are three predefined values, which are `rfc822` names, `e164` names, and `ipv4` names. If another name type is required, the type `other` may be used, where the new name type can be specified as a string of characters.

```
TransportService ::= CHOICE {
    tcpIp               [0]     NULL,
    dacapo              [1]     NULL,
    atmUni              [2]     NULL,
    mcf                 [3]     NULL,
    other               [4]     IA5String }

END
```

The `TransportService` data type determines which transport infrastructure is being used. Since this is necessary for deciding which flows may be created and which sessions a user may join, it is used in the `User` object type (described in section 2.3.1) to describe a user's binding, and the `FlowTemplate` object type (described in section 2.3.3) to specify for which transport service a flow template may be used. The predefined values of `TransportService` include `tcpIp`, `dacapo`, `atmUni`, and `mcf`. If another transport infrastructure is being used, type `other` can be used and the transport infrastructure is defined as a string of characters.

## 2.2 QoS issues

QoS parameters can be used in several ways when using GMS object types. They are used in the object types `FlowTemplate` (described in section 2.3.3) and `Flow` (described in section 2.3.4). The GMS concept of QoS parameters distinguishes a number of predefined QoS parameters and the possibility to define other QoS parameters which have one of four possible parameter types. The model of how QoS parameters are used consists of four possible steps.

The first step (which is only present if a flow template is used to create a flow) is the definition of QoS parameters within a flow template when creating the object. The second step is the creation of a flow, where the flow template's QoS parameters are used (if no flow template is used, the QoS parameters for the flow must be specified without a template). A flow's QoS parameters determine the QoS used for data transmission. The third step is the join session operation, which includes joining one or more flows. It is possible to join flows with weaker QoS parameters than defined for the flow (this only makes sense for receivers, senders must always match the flow's QoS values). It is also possible to define a weakest limit for a QoS parameter when joining the flow.

If the flow's QoS parameters have to be changed because of modified application requirements or changes in the network, the last step of QoS usage can be applied, the QoS renegotiation. QoS renegotiation may be limited by renegotiation limits (strongest and/or weakest limits may be defined). Because of the dynamic nature of GMS sessions, steps three and four may occur more than one time and may occur in any sequence.

```
GMS-QOS DEFINITIONS ::= BEGIN

QosParameterName ::= CHOICE {
    delay                       [0]     NULL,
    bandwidth                   [1]     NULL,
    peakBandwidth               [2]     NULL,
    meanBandwidth               [3]     NULL,
```

```
    minBandwidth                [4]     NULL,
    other                       [5]     IA5String }

QosParameter ::= SET {
    qosParameterName    [0]     QosParameterName,
    parameterType       [1]     CHOICE {
        unsortedValues      [0]     UnsortedList,
        sortedValues        [1]     SortedList,
        integerValues       [2]     IntegerValues,
        realValues          [3]     RealValues } OPTIONAL }
```

The QoS parameters `delay`, `bandwidth`, `peakBandwidth`, `meanBandwidth`, and `minBandwidth` are predefined values which can be used without having to introduce a new name and without having to specify their `parameterType`. All these values are of the `IntegerValues` type. If another QoS parameters should be used, the `other` type must be used and the `parameterType` must be given.

The `parameterType` of a `QosParameter` decides whether that parameter is made up of `unsortedValues`, `sortedValues`, `integerValues`, or `realValues`. The general model of QoS parameters is independent of the `parameterType`. There always is a `defaultValue` which is used for joining a flow when no local specifications of the parameter's value are specified.

```
UnsortedList ::= SET {
    values              [0]     SET OF IA5String,
    defaultValue        [1]     IA5String,
    alternatives        [2]     SET OF IA5String OPTIONAL,
    renegotiateValues   [3]     SET OF IA5String OPTIONAL }

SortedList ::= SET {
    values              [0]     SEQUENCE OF IA5String,
    defaultValue        [1]     IA5String,
    weakestLimit        [2]     IA5String OPTIONAL,
    renegotStrongLimit  [3]     IA5String OPTIONAL,
    renegotWeakLimit    [4]     IA5String OPTIONAL }
```

For `unsortedValues` and `sortedValues`, which are both just sets respectively ordered sets of strings, the QoS parameter's `values` must be defined in order to know which choices for a parameter are possible. Because no order is defined for `unsortedValues`, it is necessary to define sets of values for `alternatives` (local modifications of QoS parameters when joining the flow) and `renegotiateValues`, rather than specifying range limits. Because `sortedValues` do have an order, it is possible to define the `weakestLimit` (the limit for local QoS parameters when joining), and `renegotStrongLimit` and `renegotWeakLimit` (the limits for modifying `defaultValue` and `weakestLimit` when performing a renegotiation) as range limits of the QoS parameter's `values`.

```
IntegerValues ::= SET {
    defaultValue        [0]     INTEGER,
    weakestLimit        [1]     INTEGER OPTIONAL,
    renegotStrongLimit  [2]     INTEGER OPTIONAL,
    renegotWeakLimit    [3]     INTEGER OPTIONAL }

RealValues ::= SET {
    defaultValue        [0]     REAL,
```

```
    weakestLimit        [1]     REAL OPTIONAL,
    renegotStrongLimit  [2]     REAL OPTIONAL,
    renegotWeakLimit    [3]     REAL OPTIONAL }
```

`IntegerValues` and `RealValues` are values with a clearly defined order. Therefore, the specification of a `defaultValue` is sufficient for a parameter value. Optional components are the `weakestLimit` (the limit for local QoS parameters when joining), the `renegotStrongLimit` (the strongest limit for setting `defaultValue` and `weakestLimit` in a renegotiation), and `renegotWeakLimit` (the weakest limit for setting `defaultValue` and `weakestLimit` in a renegotiation).

```
QosParameterTempl ::= SET {
    qosParameterName    [0]     QosParameterName,
    parameterValues     [1]     CHOICE {
        unsortedValues      [0]     SET OF IA5String,
        sortedValues        [1]     SEQUENCE OF IA5String,
        integerValues       [2]     SET {
            strongestLimit      [0]     INTEGER OPTIONAL,
            weakestLimit        [1]     INTEGER OPTIONAL },
        realValues          [3]     SET {
            strongestLimit      [0]     REAL OPTIONAL,
            weakestLimit        [1]     REAL OPTIONAL } } OPTIONAL }
```

Because flow templates do not define actual data transmissions but only templates for creating them, a `QosParameterTempl` is defined differently than actual parameters. A `QosParameterTempl` is defined as a `qosParameterName` and `parameterValues`. The `parameterValues` are defined according to the four parameter types introduced above (`unsortedValues`, `sortedValues`, `integerValues`, and `realValues`). Because in a template it only makes sense to define sets of possible values (`unsortedValues` and `sortedValues`), or strongest and weakest limits for `integerValues` and `realValues`, these are the only possible specification inside a `QosParameterTempl`. This attribute type is only used in the `FlowTemplate` object type (described in section 2.3.3).

```
QosRenegotiation ::= SET {
    qosParameterName            [0]     QosParameterName,
    parameterType               [1]     CHOICE {
        unsortedValues              [0]     SET {
            defaultValue                [0]     IA5String,
            alternatives                [1]     SET OF IA5String OPTIONAL },
        sortedValues                [1]     SET {
            defaultValue                [0]     IA5String,
            weakestLimit                [1]     IA5String OPTIONAL },
        integerValues               [2]     SET {
            defaultValue                [0]     INTEGER,
            weakestLimit                [1]     INTEGER OPTIONAL },
        realValues                  [3]     SET {
            defaultValue                [0]     REAL,
            weakestLimit                [1]     REAL OPTIONAL } } }

END
```

For QoS renegotiations, which are performed using the GAP renegotiate service and the GAP renegotiation service, the `QosRenegotiation` data type is required. This data type is a set containing

a qosParameterName and renegotiation values which are defined in the parameterType. Depending on the parameterType attribute, the specification of new QoS values is defined differently, but it is always mandatory to define a new defaultValue and always optional to define a new weakestLimit respectively alternatives (if the parameter is of type unsortedValues).

### 2.3 Object types

This section describes the object types of GMS. Definitions from the previous sections (GMS-COMMON and GMS-QOS) are used to define these types. Subsections 2.3.1 to 2.3.6 contain short descriptions and the ASN.1 definitions of the object types.

```
GMS-OBJECT-TYPES DEFINITIONS ::= BEGIN

IMPORTS
    AccessRight, Application, AuthLevel, AuthRequirements, AuthType,
    CertificateType, DataType, GsaAddress, GmsObjectName, GmsObjectInformation,
    GmsRelationName, GuaAddress, NameType, TransportService
    FROM GMS-COMMON
    QosParameter, QosParameterName, QosParameterTempl
    FROM GMS-QOS;

GmsObject ::= CHOICE {
    user                [0]     User,
    group               [1]     Group,
    flowTemplate        [2]     FlowTemplate,
    flow                [3]     Flow,
    session             [4]     Session,
    certificate         [5]     Certificate }

GmsObjectAttributes ::= CHOICE {
    user                [0]     UserAttributes,
    group               [1]     GroupAttributes,
    flowTemplate        [2]     FlowTemplateAttributes,
    flow                [3]     FlowAttributes,
    session             [4]     SessionAttributes,
    certificate         [5]     CertificateAttributes }

GmsObjectRelations ::= CHOICE {
    user                [0]     UserRelations,
    group               [1]     GroupRelations,
    flowTemplate        [2]     FlowTemplateRelations,
    flow                [3]     FlowRelations,
    session             [4]     SessionRelations,
    certificate         [5]     CertificateRelations }
```

The header of the GMS-OBJECT-TYPES module imports the required data types from the modules GMS-COMMON and GMS-QOS. It also defines the data types GmsObject, GmsObjectAttributes, and GmsObjectRelations, which are used wherever generic references to GMS objects, their attributes, or relations are required. Relations are discussed in in detail section 2.4.

A general rule for the object type definitions is the separation of the definitions into an `objectName`, which always defines the object's name, `objectAttributes` (which may be modified by a sufficiently authorized user), `objectInternalAttributes` (which may only be modified by the GMS itself), and `objectRelations` (which describe the relations that an object may have). The `objectInternalAttributes` are not present in all object type definitions.

### 2.3.1 User

A `User` object is used to represent a GMS user, ie a real world person. Because authentication and authorization of the GMS heavily depend on the concept of `User` objects, it is one of the most important object types. And because application needs in terms of security differ very much, authentication and authorization can be handled very flexible.

```
User ::= --snacc isPdu:"TRUE" -- SET {
    userName            [0]     CHOICE {
        name                [0]     GmsObjectName,
        anonymous           [1]     SET {
            identifier          [0]     INTEGER,
            gsaAddress          [1]     GsaAddress } },
    userAttributes      [1]     UserAttributes,
    userIntAttributes   [2]     UserInternalAttributes,
    userRelations       [3]     UserRelations }
```

A `User` object may either have a name or refer to an `anonymous` GMS user. In this case, the object is not referenced with a `GmsObjectName` but with a combination of the `gsaAddress`, pointing to the GSA the anonymous user bound to, and an `identifier` for referring to the anonymous user. For the remainder of the object type definition, the `User` attributes are according to the general rules for the object type definitions.

```
UserAttributes ::= SET {
    realWorldName       [0]     IA5String,
    userInformation     [1]     GmsObjectInformation OPTIONAL,
    userMailAdress      [2]     IA5String OPTIONAL,
    authRequirements    [3]     AuthRequirements,
    authInformation     [4]     SET OF SET {
        authType            [0]     AuthType,
        authInformation     [1]     OCTET STRING },
    userAccessPolicy    [5]     SET {
        owner               [0]     AccessRight,
        members             [1]     AccessRight,
        world               [2]     AccessRight } }
```

The `UserAttributes` include all information which is necessary for identification and authentication of a user. The `realWorldName` simply is a string of characters defining the real world name of a user. The `userInformation` is the standard optional `GmsObjectInformation` which may be stored with each GMS object. Because it may be useful to be able to contact a GMS user asynchronously via email, the `userMailAdress` is defined, but it is optional. The `authRequirements` of a user (defined and described in section 2.1) decide, which authentication level a user must have before he is allowed to perform any of the actions the `userAccessPolicy` defines. The `authInformation` of a user is a set of possible authentication methods. The user may choose different methods depending on the

authentication requirements of the objects he is going to access. The `userAccessPolicy` defines the access rights for the user object. It uses the `AccessRight` definition from the `GMS-COMMON` module. The `members` keyword in this case defines the access rights for users which are in the same group as the user being accessed.

```
UserInternalAttributes ::= SET {
    bindings            [0]     SET OF SET {
        address             [0]     GuaAddress,
        authentication      [1]     AuthType,
        transportServices   [2]     SET OF TransportService,
        bindingComment      [3]     IA5String OPTIONAL,
        applications        [4]     SET OF Application OPTIONAL } OPTIONAL }
```

The `UserInternalAttributes` contain the bindings of a user, ie the information on where the user is currently actively working with the GMS. Each binding consists of an `address`, describing where (ie at which GSA and which GUA address at this GSA) the user is bound, `authentication` information describing which authentication method has been used when creating this binding, and a set of `transportServices`, describing which transport services are available with the communication infrastructure which has initiated the binding. Two optional component of a binding are a `bindingComment`, which is a description of the binding given as a string of characters, and `applications`, which is a set of `Application` identifiers describing which applications the user is using for this binding.

```
UserRelations ::= SET {
    owns                [0]     SET OF GmsRelationName OPTIONAL,
    manages             [1]     SET OF GmsRelationName OPTIONAL,
    member              [2]     SET OF GmsRelationName OPTIONAL,
    participant         [3]     SET OF GmsRelationName OPTIONAL,
    sends               [4]     SET OF GmsRelationName OPTIONAL,
    receives            [5]     SET OF GmsRelationName OPTIONAL }
```

The `UserRelations` of a user object describe the relations which may be established between a user and other objects. The `owns` relation describes for which objects the user is the owner, which gives him special access rights for these objects. By definition, each user is the owner of his user object (because of this definition, the user object type does not have an owner relation defined, which is present in all other object types). The `manages` relation specifies for which objects the user is a manager. The status as a manager gives special access privileges for objects. Being a manager for a group or session also may require to response to manager requests which are necessary for approving or refusing join group or join session requests.

`member` and `participant` relations are established when a user joins a group (becoming a group member) or a session (becoming a session participant). The relations are deleted when the group or session is left. The `sends` and `receives` relations are used for establishing relationships between users and flows, whenever a user joins a flow (by joining a session) as a sender and/or receiver. These relations are also deleted when the session the flows are part of is left.

### 2.3.2 Group

A `Group` object is used to refer to a group of users, which are called group members. However, because groups may be nested (group can be members of group), group membership of a user can be either direct or indirect. Groups are the central instrument for granting authorizations, because

the authorization check for session joining can be based on group membership. Groups may also be used for administrative purposes (such as creating groups which may be used for electronic mail), they represent the static abstraction for grouping users, while sessions represent the more dynamic abstraction (ie communications).

```
Group ::= --snacc isPdu:"TRUE" -- SET {
    groupName           [0]     GmsObjectName,
    groupAttributes     [1]     GroupAttributes,
    groupRelations      [2]     GroupRelations }

GroupAttributes ::= SET {
    realWorldName       [0]     IA5String,
    groupInformation    [1]     GmsObjectInformation OPTIONAL,
    groupMailAddress    [2]     IA5String OPTIONAL,
    authRequirements    [3]     AuthRequirements,
    groupAccessPolicy   [4]     SET {
        owner               [0]     AccessRight,
        managers            [1]     AccessRight,
        members             [2]     AccessRight,
        world               [3]     AccessRight },
    staticGroup         [5]     BOOLEAN,
    groupMembers        [6]     ENUMERATED {
        users               (1),
        groups              (2),
        usersAndGroups      (3) },
    groupJoinPolicy     [7]     CHOICE {
        relativeQuorum      [0]     INTEGER (0..100),
        absoluteQuorum      [1]     INTEGER },
    groupNotificaPolicy [8]     SET {
        joinGroup           [0]     GroupNotification,
        leaveGroup          [1]     GroupNotification,
        bind                [2]     GroupNotification,
        unbind              [3]     GroupNotification,
        createSession       [4]     GroupNotification,
        deleteSession       [5]     GroupNotification } }

GroupNotification ::= ENUMERATED {
    none                (1),
    managers            (2),
    members             (3),
    managersAndMembers  (4) }
```

The `GroupAttributes` contain all information which define a group's properties. The `realWorldName` is a string of characters which describes the group with a short name or explanation. `groupInformation` is the optional standard information associated with every GMS object. The `groupMailAddress` is also defined as a string of characters and contains the group's mail address, which is optional. This attribute may be useful if information has to be sent to all group members asynchronously using electronic mail. However, to keep the GMS group's members and the mail group's members consistent entirely depends on the maintainer of the mail group.

The authRequirements of a group (defined and described in section 2.1) decide which authentication level a user must have before he is allowed to perform any of the actions the groupAccessPolicy defines. If the group is a staticGroup, then join and leave operations are not allowed (ie the set of members is static). The groupMembers attribute is used to decide whether only users, only groups, or usersAndGroups are allowed to join the group. Thus, the groupMembers attribute can be used to control the group hierarchy.

The groupJoinPolicy determines how join group requests are processed. The relativeQuorum policy defines the percentage of managers which must at least confirm to approve the join group request. Setting this parameter to zero creates an entirely open group, setting it to hundred creates a group where all managers must confirm for a join to be successful. However, it should be kept in mind that it is not very likely that all managers will always be bound to the GMS. It is also possible to specify an absoluteQuorum which gives the number of managers which must at least confirm to approve the join request. If a group should be open for joins without the need to request managers at all, either the relativeQuorum or the absoluteQuorum can be set to zero.

The groupNotificaPolicy defines the policy which is used for notifying group members and/or managers of certain events. For every event it is possible to choose a GroupNotification type, which decides whether nobody, only managers, only members, or managersAndMembers should be notified. Notifiable events are successful joinGroup and leaveGroup requests, which change the group's member set (it is important to notice that only joins and leaves of direct members are notified), bind and unbind requests of group members (which change the active population), and createSession and deleteSession requests, which change the ongoing communications which are associated with this group.

```
GroupRelations ::= SET {
    owners              [0]     GmsRelationName,
    managers            [1]     GmsRelationName OPTIONAL,
    member              [2]     SET OF GmsRelationName OPTIONAL,
    members             [3]     GmsRelationName OPTIONAL,
    sessions            [4]     GmsRelationName OPTIONAL }
```

The GroupRelations of a group object describe the relations which may be established between a group and other objects. The owners relation is used to determine which users are owners of the group. The managers and the members relations define the managers and members of the group. The sessions relation describes which sessions are associated with the group. The member relation describes, of which groups this group is a member of.

### 2.3.3 Flow template

A FlowTemplate object describes a template which may be used to create a flow. However, it is not necessary that a flow is based on a template. It is possible to store information about a flow, its construction and its properties (QoS parameters) inside a FlowTemplate object. When a flow is created based on the flow template's content, all data from the template can be used to create the flow. Depending on the actual communication infrastructure being used, this may save a lot of work (ie, protocol configurations may be predefined which do not have to be computed if they are stored inside a flow template)

```
FlowTemplate ::= --snacc isPdu:"TRUE" -- SET {
    flowTemplateName    [0]     GmsObjectName,
    flowTemplAttributes [1]     FlowTemplateAttributes,
    flowTemplRelations  [2]     FlowTemplateRelations }
```

```
FlowTemplateAttributes ::= SET {
    flowTemplInfo       [0]     GmsObjectInformation OPTIONAL,
    dataType            [1]     DataType,
    transportService    [2]     TransportService,
    flowTemplateData    [3]     OCTET STRING OPTIONAL,
    flowTemplDirection  [4]     ENUMERATED {
        uniDirectional      (1),
        biDirectional       (2) } OPTIONAL,
    participantLimits   [5]     SEQUENCE {
        maxnumSenders       [0]     INTEGER OPTIONAL,
        maxnumReceivers     [1]     INTEGER OPTIONAL } OPTIONAL,
    qosParameterTempl   [6]     SET OF QosParameterTempl }
```

A `FlowTemplate` is defined by its name (the `flowTemplateName`), `FlowTemplateAttributes`, and `FlowTemplateRelations`. The `FlowTemplateAttributes` contain all information which is necessary to describe a `FlowTemplate` object. `flowTemplInfo` is the optional standard information associated with every GMS object. The `DataType` of a `FlowTemplate` is described in section 2.1, it describes which type of data may be transported with a flow which is based on this `FlowTemplate`. The `TransportService` is also described in section 2.1, it describes which transport infrastructure is necessary to create a flow based on this `FlowTemplate`.

The `flowTemplateData` is data which is necessary to create a flow based on the `FlowTemplate`. This data is completely dependent on the communication infrastructure for which this `FlowTemplate` has been created. Additional information about the flow template includes the possible direction modes (`flowTemplDirection` may be `uniDirectional` or `biDirectional`). The `participantLimits` may be given to specify a maximum number of senders and/or receivers for a flow based on this `FlowTemplate`. It is possible that an actual flow which is based on this template has different values than the ones defined in the template. This depends on how the flow creation is done.

The `qosParameterTempl` is a set of `QosParameterTempl`, which are described in more detail in section 2.2. Using a flow template's `qosParameterTempl` it is possible to specify the properties of a flow which is created based on this `FlowTemplate`. Because the template is generic, it is very likely that the QoS parameters of the template are not fully specified in the template. However, the `qosParameterTempl` information is very useful in determining whether a flow to be created can be based on this `FlowTemplate` or not.

```
FlowTemplateRelations ::= SET {
    owners              [0]     GmsRelationName }
```

The `FlowTemplateRelations` are very simple, because a `FlowTemplate` has only the `owners` relation which specifies which users are the owners of this object.

### 2.3.4 Flow

A `Flow` object is the GMS representation of one flow of data between several GMS users. The GMS model of data exchange between users is based on sessions (described in section 2.3.5) and flows. Each session consists of one or more flows, which are said to be part of the session. It is possible to specify dependencies and synchronization relations between the flows of a session. Depending on the application, different mappings from application requirements onto sessions and flows are possible. It is up to the application programmer to decide which mapping onto sessions and flows he chooses.

```
Flow ::= --snacc isPdu:"TRUE" -- SET {
    flowName            [0]     GmsObjectName,
    flowAttributes      [1]     FlowAttributes,
    flowIntAttributes   [2]     FlowInternalAttributes,
    flowRelations       [3]     FlowRelations }

FlowAttributes ::= SET {
    flowInformation     [0]     GmsObjectInformation OPTIONAL,
    flowData            [1]     OCTET STRING OPTIONAL,
    flowDirection       [2]     ENUMERATED {
        uniDirectional      (1),
        biDirectional       (2) },
    participantLimits   [3]     SEQUENCE {
        maxnumSenders       [0]     INTEGER OPTIONAL,
        maxnumReceivers     [1]     INTEGER OPTIONAL } OPTIONAL,
    renegotiation       [4]     SET {
        senders             [0]     BOOLEAN,
        receivers           [1]     BOOLEAN,
        sessionManagersOnly [2]     BOOLEAN } OPTIONAL }
```

A `Flow` consists of its name (the `flowName`), `flowAttributes`, `flowIntAttributes`, and `flowRelations`. The `FlowAttributes` contain the optional standard information associated with every GMS object. `flowData` is the application specific data which is necessary to actually create the flow when joining the session the flow is part of. This information may be empty when addressing information is sufficient for joining the flow, but it may also contain data structures which are necessary for setting up the protocol which is being used with that flow.

Depending on whether the flow is `uniDirectional` or `biDirectional`, the `flowDirection` attribute is set accordingly. The `participantLimits` of a flow specify, if present, a `maxnumSenders` and/or a `maxnumReceivers`, which define the maximum number of participants which may send data to or receive data from this flow. If the flow is `biDirectional`, it may still be possible to only join as a receiver (even though no more senders are accepted), so the distinction between `maxnumSenders` and `maxnumReceivers` may also be useful in this case.

The optional `renegotiation` attribute decides who is allowed to initiate a QoS renegotiation for this flow. The QoS are described in greater detail in section 2.2. Depending on the `renegotiation`, `senders` or `receivers` are allowed to initiate a QoS renegotiation (ie to create a GAP renegotiation request). If `sessionManagersOnly` is set to true, only session managers are allowed to initiate a QoS renegotiation. If all `renegotiation` attributes are set to false, the flow is not renegotiable.

```
FlowInternalAttributes ::= SET {
    qosParameters       [0]     SET OF QosParameter,
    addressingInfo      [1]     CHOICE {
        receiverOriented    [0]     OCTET STRING,
        senderOriented      [1]     SET OF SEQUENCE {
            userName            [0]     GmsObjectName,
            address             [1]     OCTET STRING } }
 }
```

The `FlowInternalAttributes` of a flow include information about QoS parameters and addressing information. The `qosParameters` attribute of a flow describes all properties of the flow which are defined by QoS parameters. Section 2.2 contains a detailed description of the GMS QoS concept. The

`addressingInfo` of a flow contains information which is necessary for joining the flow. If the flow is based on `receiverOriented` addressing, then one address (the flow address, sometimes called group address) is sufficient for joining the flow. If the flow is `senderOriented` addressed, each newcomer must get the complete set of addresses of flow receivers. Therefore, in this case the `addressingInfo` is a set of addresses which may be used for addressing all receivers of the flow. Each member of this set consists of the receiver's name (the `userName`) and the `address` the receiver is using. This set is automatically updated when receivers join or leave the flow.

```
FlowRelations ::= SET {
    session             [0]     GmsRelationName,
    senders             [1]     GmsRelationName,
    receivers           [2]     GmsRelationName,
    dependencies        [3]     SET OF GmsRelationName OPTIONAL,
    synchronized        [4]     SET OF GmsRelationName OPTIONAL }
```

The `FlowRelations` define the relations which may be established between a flow object and other objects. The `session` relation describes which session this flow is a part of. `senders` and `receivers` are relations which are used to register the users which are senders and/or receivers of the flow. The `dependencies` of a `Flow` are directed relations which specify whether this flow depends on another flow or whether another flow depends on this flow. The concept of flow dependencies is described in greater detail in section 2.4.2. `synchronized` flows may be achieved by establishing this relation. The concept of synchronized flows is described in greater detail in section 2.4.10.

### 2.3.5 Session

A `Session` object is the main abstraction of the GMS for communications between users. Each `Session` consists of one or more flows, which are the abstractions for the actual data exchange connections. However, a `Session` is one manageable unit which is used for authentication, authorization, admission control, and management. It is only possible to participate in communications by using sessions, flows are always joined and left by joining and leaving sessions. Therefore, the concept of sessions and flows is the central concept of communication inside the GMS.

```
Session ::= --snacc isPdu:"TRUE" -- SET {
    sessionName         [0]     GmsObjectName,
    sessionAttributes   [1]     SessionAttributes,
    sessionInAttributes [2]     SessionInternalAttributes,
    sessionRelations    [3]     SessionRelations }

SessionAttributes ::= SET {
    sessionInformation  [0]     GmsObjectInformation OPTIONAL,
    sessionAppInfo      [1]     SET {
        application         [0]     Application,
        appSpecificInfo     [1]     OCTET STRING } OPTIONAL,
    sessionDuration     [2]     SEQUENCE {
        start               [0]     UTCTime OPTIONAL,
        end                 [1]     UTCTime OPTIONAL } OPTIONAL,
    blockingJoinLeave   [3]     BOOLEAN,
    anonymousSession    [4]     BOOLEAN,
    authRequirements    [5]     AuthRequirements,
    sessionAccessPolicy [6]     SET {
```

```
        owner               [0]     AccessRight,
        managers            [1]     AccessRight,
        participants        [2]     AccessRight,
        world               [3]     AccessRight },
    sessionJoinPolicy   [7]     CHOICE {
        open                [0]     NULL,
        group               [1]     NULL,
        managed             [2]     CHOICE {
            relativeQuorum      [0]     INTEGER (1..100),
            absoluteQuorum      [1]     INTEGER } },
    sessionNotifiPolicy [8]     SET {
        joinSession         [0]     SessionNotification,
        leaveSession        [1]     SessionNotification } }

SessionNotification ::= ENUMERATED {
    none                (1),
    managers            (2),
    participants        (3),
    managersAndParticip (4) }
```

A `Session` object consists of its `sessionName`, `sessionAttributes`, internal `sessionInAttributes`, and `sessionRelations`. `sessionInformation` is the optional standard information associated with every GMS object. The `sessionAppInfo` is application specific information which is also optional. Each member of the set of `sessionAppInfo` consists of an `application` identification and `appSpecificInfo`, which may be any information that an application needs to associate with this session.

An optional `sessionDuration` can be given to specify the session's lifetime. It is possible to specify the `start` and/or the `end` of the session. Using this information it is possible to define sessions although they are not yet active.

The `blockingJoinLeave` attribute decides whether join and leave operations for this session are blocking. Blocking join and leave operations cause a serialization of join and leave requests. This makes it possible to ensure that participant limits of flows are never exceeded. Consequently, a session which contains flows with participant limits should set the `blockingJoinLeave` attribute to true. If no such flow is present in a session, this setting might slow down the join and leave operation unnecessarily.

An `anonymousSession` is a session where the identity of the senders and receivers should not be given to users. Because the identity must be known to GUAs (in order to make it possible to set up connections), it is not possible to keep the information about senders and receivers inside the GSAs only. However, the communication frameworks containing the GUAs should not give any information about session participants to users if the session is an `anonymousSession`.

The `authRequirements` of a session (defined and described in section 2.1) decide which authentication level a user must have before he is allowed to perform any of the actions the `sessionAccessPolicy` defines. The `sessionJoinPolicy` describes how join requests are handled. If the `sessionJoinPolicy` is `open`, then everybody may join the session. If the `sessionJoinPolicy` is `group`, then only members of the group the session is associated with are allowed to join the session (a description of the association of sessions and groups is given in section 2.4.1). If the `sessionJoinPolicy` is set to `managed`, then a mechanism identical to the one described for managed groups (section 2.3.2) is used.

The `sessionNotifiPolicy` decides whether session managers and/or participants should be notified of new members and/or members leaving the session. It is possible to enable these notifications

for join and leave operations separately. It can be chosen whether managers, participants, or managers and participants (managersAndParticip) should be notified.

```
SessionInternalAttributes ::= SET {
    joinLeaveRequests   [0]     SEQUENCE OF SET {
        userName            [0]     GmsObjectName,
        pendingRequest      [1]     ENUMERATED {
            join                (1),
            leave               (2) } } OPTIONAL }
```

A session's SessionInternalAttributes contain the pending joinLeaveRequests of a session. These requests consist of a userName and an identification of the pendingRequest, which can be a join or a leave request. This attribute is used when the blockingJoinLeave attribute is set to true and thus join and leave requests are serialized.

```
SessionRelations ::= SET {
    owners              [0]     GmsRelationName,
    managers            [1]     GmsRelationName OPTIONAL,
    participants        [2]     GmsRelationName OPTIONAL,
    flows               [3]     GmsRelationName OPTIONAL,
    group               [4]     GmsRelationName OPTIONAL }
```

The SessionRelations define which relations may be established between a session object and other GMS objects. The owner of a session (the owner relation is described in detail in section 2.4.5) has special rights which are defined in the owners part of the sessionAccessPolicy. The managers of a group also have special access rights as well as maybe a different notification policy set in the sessionNotifiPolicy than normal participants. A sessions participants are the users actively taking part in the communication going on inside the session using its flows. The flows itself may have relations established among themselves, which is described in section 2.3.4. Finally, a session's group describes the group to which the session is associated. This is especially important and a mandatory relation if the sessionJoinPolicy is set to group.

### 2.3.6 Certificate

A Certificate object is used to store a certificate which may be used to prove the authenticity of another object. This may be used to store information about objects which need to be authentic in order to be able to guarantee some level of security. An example for such an application of the Certificate object is the certification of software components which will be used for exchanging security sensitive data. Only if it possible to check whether the software being used is authentic secure data exchange can be guaranteed.

```
Certificate ::= --snacc isPdu:"TRUE" -- SET {
    certificateName     [0]     GmsObjectName,
    certificAttributes  [1]     CertificateAttributes,
    certificRelations   [2]     CertificateRelations }

CertificateAttributes ::= SET {
    certificInformation [0]     GmsObjectInformation OPTIONAL,
    certificateType     [1]     CertificateType,
    nameType            [2]     NameType,
    validity            [3]     SEQUENCE {
```

```
        notbefore           [0]     UTCTime,
        notafter            [1]     UTCTime },
    name                [4]     IA5String,
    data                [5]     BIT STRING,
    signatures          [6]     BIT STRING }
```

A `Certificate` consists of the `certificateName`, `certificAttributes`, and `certificRelations`. The `CertificateAttributes` contain the optional standard information associated with every GMS object (`certificInformation`). The `certificateType` and the `nameType` contain information about the certificate which is described in section 2.1. The `validity` of a certificate is given as a sequence of times, the first indicating the earliest validity (`notbefore`), the second specifying the end of the validity period (`notafter`).

The certificate's `name` is a string according to the certificate's `nameType`. The `data` and `signatures` is information necessary for the actual certification process and both of them are given as a bit string.

```
CertificateRelations ::= SET {
    owners              [0]     GmsRelationName }

END
```

The only relation a certificate may have is the `owners` of the certificate. Only the owner of a certificate is allowed to modify it or to remove it from the GMS.

## 2.4 Relations

Relations are defined as possible relationships between two or more GMS objects. The relations described in this sections are all used within some of the `GmsObjectRelations` described in section 2.3. For every relation it will be described objects of which object types may be used with it. With the two exceptions of the dependency and the synchronization relation (which are described in section 2.4.2i and 2.4.10), all relations are defined as a set containing a relation name, a reference to one object, and a set of references to other objects, which are all related to the object referred to in the second attribute of the relation.

```
GMS-RELATIONS DEFINITIONS ::= BEGIN

IMPORTS
    GmsObjectName, GmsRelationName
    FROM GMS-COMMON;

GmsRelation ::= CHOICE {
    association                 [0]     Association,
    dependency                  [1]     Dependency,
    manager                     [2]     Manager,
    member                      [3]     Member,
    owner                       [4]     Owner,
    part                        [5]     Part,
    participation               [6]     Participation,
    receiver                    [7]     Receiver,
    sender                      [8]     Sender,
    synchronization             [9]     Synchronization }
```

The `GmsRelation` type is defined to make a simple reference to GMS relations possible by just using this type. It also lists all relation types available, which are described in detail in the following sections.

### 2.4.1 Association

An `Association` relation describes the relation between a group and a session. A session may be associated to a group. This relation is a 1:n relation, since a number of sessions can be associated to one group, but each session can be associated with at most one group. As long as sessions are associated to a group, it is not possible to delete this group using the GAP delete service.

```
Association ::= --snacc isPdu:"TRUE" -- SET {
    associationName             [0]     GmsRelationName,
    group                       [1]     GmsObjectName,
    session                     [2]     SET OF GmsObjectName }
```

The usage of an `Association` is to express the relationship between a group and a session. This may be used just for informal purposes. However, if the session's `sessionJoinPolicy` is set to `group` (which is described in detail in section 2.3.5), the handling of join requests for that session is based on the session's associated group and the fact whether the join requester is a member of that group. Only if he is, the join will be accepted.

### 2.4.2 Dependency

The `Dependency` is a relation which is established between a number of flows. It is used to express the fact that a flow depends on another flow, ie a flow can only be interpreted correctly (according to the creator of the session with the dependent flows), if the flow it depends on is also present. The most popular example for this is MPEG-2 coding as standardized by ISO [9], which uses data partitioning for video streams (sending low quality video images on one flow and an an additional flow which contains more information to get a video with higher quality) and multi-channel audio (which may be used for stereophonic audio signals).

```
Dependency ::= --snacc isPdu:"TRUE" -- SET {
    dependencyName              [0]     GmsRelationName,
    flows                       [1]     SEQUENCE OF GmsObjectName }
```

Each `Dependency` is a sequence of references to flows, which has to be interpreted as a directed dependency graph from the first flow to the last flow, ie the first flow depends on all subsequent flows referred to in the relation. As a consequence, only the last flow of a `Dependency` can be interpreted independently from all other flows.

It is also possible that a flow occurs in several `Dependency` relations, which is reflected by the fact that the `dependencies` attribute in the flow's `FlowRelations` (which is described in section 2.3.4) is defined to be a `SET OF GmsRelationName`. For example, the usage of multiple dependencies is required if it is necessary that a video stream depends on a multi-channel coded audio stream. In this setup, the first `Dependency` is defined between the audio channels, while the second `Dependency` establishes a relationship between the video and the audio.

### 2.4.3 Manager

The `Manager` relation establishes a relationship between a GMS object and one or more users. Groups and sessions are the only GMS objects which may have managers. The manager status

gives special access rights, which are defined in the groupAccessPolicy and sessionAccessPolicy, respectively. Managers also have special responsibilities for granting or refusing join requests to groups or sessions, if the groupJoinPolicy or sessionJoinPolicy is set accordingly. A last special thing about managers is their special status with respect to notifications, which are defined in the groupNotificaPolicy and the sessionNotifiPolicy. Details for these procedures and mechanisms are given in section 2.3.2 for groups and section 2.3.5 for sessions.

```
Manager ::= --snacc isPdu:"TRUE" -- SET {
    managerName                 [0]     GmsRelationName,
    managedObject               [1]     GmsObjectName,
    users                       [2]     SET OF GmsObjectName }
```

Because groups or sessions may have several managers (which may be added or deleted using the GAP modify service), the users part of the Manager relation is defined to be a SET OF GmsObjectName. Each group or session object only has one set of managers related to it, which is why the managers attribute in the GroupRelations and SessionRelations is defined as a GmsRelationName. On the other hand, a user may be manager of several objects, which is why the specification of the user object type (which is given in section 2.3.1) defines the manages attribute of the UserRelations to be a SET OF GmsRelationName.

### 2.4.4 Member

The Member relation is used to express the membership of users within groups. Consequently it is modified whenever a join or leave request for a group has been successful. Because groups may have more than one member, the relation is a 1:n relation. If a group is defined to be a staticGroup (described in section 2.3.2), the Member relation will not change, because the members of the group may not be changed.

```
Member ::= --snacc isPdu:"TRUE" -- SET {
    memberName                  [0]     GmsRelationName,
    group                       [1]     GmsObjectName,
    members                     [2]     SET OF GmsObjectName }
```

Because users and groups may be members of a group (this depends on how the groupMembers attribute of the group is set), the members of a group may be references to objects of these two types. Each group only has one set of members, therefore the members attribute in the definition of the GroupRelations is only a GmsRelationName. However, because users and groups may be members of multiple groups, the member attribute of the UserRelations and GroupRelations, respectively, is defined to be a SET OF GmsRelationName.

### 2.4.5 Owner

The Owner relation is used to define owners for GMS objects. Groups, flow templates, sessions, and certificates have owners. Groups and sessions have a groupAccessPolicy or a sessionAccessPolicy which defines the AccessRight of the owners of these objects. Flow templates and certificates are by definition readable by everyone and may only be modified or deleted by their owners, so they have no explicit access rights defined.

```
Owner ::= --snacc isPdu:"TRUE" -- SET {
    ownerName                   [0]     GmsRelationName,
    ownedObject                 [1]     GmsObjectName,
    owners                      [2]     SET OF GmsObjectName }
```

Because each object which has owner may have multiple owners (which may be added or deleted using the GAP modify service), the `owners` attribute inside the `Owner` relation is defined to be a `SET OF GmsObjectName`. An owner always is a user. Because each object may only have one set of owners, the `owners` attribute is a `GmsRelationName`. Because each user may be owner of several GMS objects, the `owns` attribute of the user's `UserRelations` is a defined to be a `SET OF GmsRelationName`.

### 2.4.6 Part

The `Part` relation describes the relationship between a session and a number of flows. By definition (as described in sections 2.3.4 and 2.3.5), each flow belongs to a session. It is created when the session is created and deleted when the session is deleted. When the flow is created, the part relation is established and it is not changed until the session and the flows which are part of it is deleted (using the GAP delete service).

```
Part ::= --snacc isPdu:"TRUE" -- SET {
    partName                    [0]     GmsRelationName,
    session                     [1]     GmsObjectName,
    flows                       [2]     SET OF GmsObjectName }
```

Each session can have several flows to be part of the session, so the `flows` attribute of the `Part` relation is defined to be a `SET OF GmsObjectName`. Each session only has one set of flows which are part of it, so the `flows` attribute of the session's `SessionRelations` (as described in section 2.3.5) is a `GmsRelationName`. And because every flow is part of exactly one session, the `session` attribute of the flow's `FlowRelations` (as described in section 2.3.4) also is a `GmsRelationName`.

### 2.4.7 Participation

The `Participation` relation is used to establish the relationship between a session and the users who successfully joined the session. Consequently it is modified whenever a join or leave request for a session has been successful. The participation in a session has several effects, which are described in section 2.3.5. For example, the participants of a session receive notifications about certain events, depending on the session's `sessionNotifiPolicy` (this attribute may be set to deliver join and/or leave notifications to session participants).

```
Participation ::= --snacc isPdu:"TRUE" -- SET {
    participationName           [0]     GmsRelationName,
    session                     [1]     GmsObjectName,
    users                       [2]     SET OF GmsObjectName }
```

Each session has one set of participants, which is why the `users` attribute of the `Participation` relation is defined to be a `SET OF GmsObjectName`. Because there is only one such set for every session, the `participants` attribute in the session's `SessionRelations` is defined to be a `GmsRelationName`. However, because every user may be participant of several sessions, the `participant` attribute of the user's `UserRelations` is defined to be a `SET OF GmsRelationName`.

### 2.4.8 Receiver

The `Receiver` relation is used to specify the relationship between a flow and all receiving users for that flow. When joining a session, a user also has to specify which flows he wants to join using which role. The `Receiver` relation is especially important for dealing with renegotiations, because all receivers of a flow must get a renegotiation notification when a QoS renegotiation for this flow is initiated using the GAP renegotiate service.

```
Receiver ::= --snacc isPdu:"TRUE" -- SET {
    receiverName                [0]     GmsRelationName,
    flow                        [1]     GmsObjectName,
    users                       [2]     SET OF GmsObjectName }
```

Because each flow may have a number of receivers, the `users` attribute of the `Receiver` relation is defined to be a `SET OF GmsObjectName`. However, because each flow only has one receiver set, the `receivers` attribute of the flow's `FlowRelations` (described in section 2.3.4) is defined to be a `GmsRelationName`. On the other hand, a user may be receiver for more than one flow, therefore the `receives` attribute of the user's `UserRelations` (described in section 2.3.1) is defined as a `SET OF GmsRelationName`.

### 2.4.9 Sender

The `Sender` relation is used to specify the relationship between a flow and all sending users for that flow. When joining a session, a user also has to specify which flows he wants to join using which role. The `Sender` relation is especially important for dealing with renegotiations, because all senders of a flow must get a renegotiation notification when a QoS renegotiation for this flow is initiated using the GAP renegotiate service.

```
Sender ::= --snacc isPdu:"TRUE" -- SET {
    senderName                  [0]     GmsRelationName,
    flow                        [1]     GmsObjectName,
    users                       [2]     SET OF GmsObjectName }
```

Because each flow may have a number of senders, the `users` attribute of the `Sender` relation is defined to be a `SET OF GmsObjectName`. However, because each flow only has one sender set, the `senders` attribute of the flow's `FlowRelations` (described in section 2.3.4) is defined to be a `GmsRelationName`. On the other hand, a user may be sender for more than one flow, therefore the `sends` attribute of the user's `UserRelations` (described in section 2.3.1) is defined as a `SET OF GmsRelationName`.

### 2.4.10 Synchronization

The `Synchronization` relation is established between a number of flows. It is used to express the fact that this set of flows is to be synchronized. The actual implementation of the flows' synchronization is application specific and can not be defined inside the GMS. The most popular example for flows to be synchronized is the separate transmission of audio and video data using two flows, which must then be synchronized at the receiver side to achieve lip synchronization. Another example for synchronization would be the synchronization of two audio channels for playback of a stereophonic signal. More examples for possible synchronization scenarios can be found in a book by Steinmetz and Nahrstedt [19].

```
Synchronization ::= --snacc isPdu:"TRUE" -- SET {
    synchronizationName         [0]     GmsRelationName,
    flows                       [1]     SET OF GmsObjectName }

END
```

Each `Synchronization` is a set of references to flows, which is interpreted as the set of flows which have to be synchronized. Because flows may only be joined by joining sessions, all flows of the

set of a `Synchronization` relation must be parts of one session. Because a flow may be synchronized with multiple sets of other flows, the `synchronized` attribute in a flow's `FlowRelations` is defined as a `SET OF GmsRelationName`.

# 3 GSP Procedures

GSP is specified in two parts, one defining the behavior of the communicating entities, the other defining the data being exchanged. The behavior is defined in this section using textual descriptions, state diagrams, and time sequence diagrams. The exchanged data is defined in ASN.1 PDUs, which are listed and explained in section 4. Both sections together form a complete description of GSP. The semantics of each operation is described in the corresponding subsection of section 4.

The rest of this section describes several procedural aspects of GSP. The grouping of GSAs in domains and their organization are described in section 3.1. Section 3.2 contains a descriptions of the underlying protocols and their properties and usage for the transmission of PDUs. In section 3.4, it is described how GSAs are inserted to and deleted from domains. Section 3.3 describes the dissemination of domain information inside domains which is used to keep track of status changes of GSAs in all GSAs of a domain. The concept of tokens is the subject of section 3.5, which also discusses the various types and usages of GSP tokens. Section 3.6 finally describes the processing of GAP initiated operations in GSP.

## 3.1 Domains

GSAs are grouped into domains, each consisting of a number of GSAs. Domains are organized hierarchically, ie normally every domain has one superdomain and a number of subdomains. However, there is no such thing as a root domain but a set of top-level domains. An example for such a structure is depicted in figure 2.

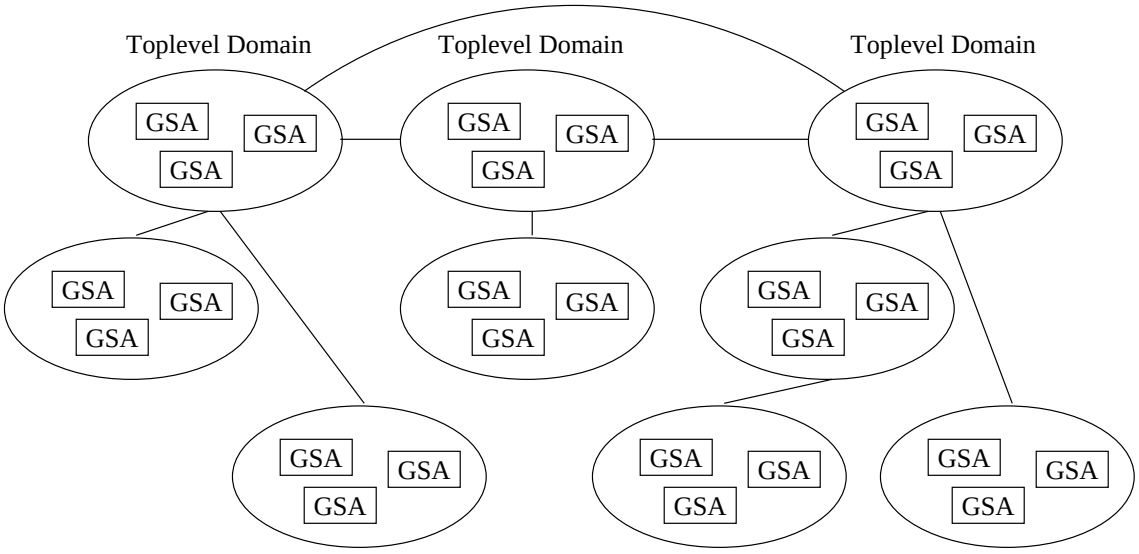


Figure 2: GMS domain hierarchy

In this figure, domains connected by solid lines know each others address (through GSA configurations, which normally take place when a GSA is started), whereas all other domains do not know each other. This way, the knowledge about the complete domain hierarchy is distributed (ie there is no GSA which initially knows all GMS domains), and new domains can be added or removed locally without having to inform all GSAs. The GSP part which enables GSAs to learn about domains

which are not immediate neighbors is described in section 3.6.1. However, it depends on a GSAs implementation whether a domain's address is evaluated every time something has to be sent to the domain or whether these addresses are cached and reused.

## 3.2 Underlying protocols

GSP uses two different transport infrastructures for transmitting data. The first infrastructure is a reliable, FIFO ordered (according to the definitions given by Hadzilacos and Toueg [8]) multipoint-to-multipoint transport infrastructure. Currently, the multicast protocol described by Bauer et al. [1] and implemented by Nigg [15] is being used. The second infrastructure GSP uses is a reliable, connection-oriented point-to-point transport infrastructure. Currently, TCP/IP (described for example by Stevens [20]) is being used.

The multicast infrastructure is used in all cases where it is necessary to distribute a PDU to all GSAs of a domain. Consequently, each domain has its own multicast address. This address must be assigned when creating a new domain a configured into all GSAs of the new domain and all neighboring domains. Section 3.6.1 describes how the address of the target domain is evaluated, if a PDU needs to be sent to this domain. PDUs sent by multicast are `JoinDomainRequest` (described in section 4.1.1), `DomainInfo` (described in section 4.1.2), `LeaveDomainRequest` (described in section 4.1.3), `DomainResolveRequest` (described in section 4.1.4), `InitToken` and `ClaimToken` (described in section 4.2.1), `IdentifyTHRequest` (described in section 4.2.2), `AddRel2ObjRequest` (described in section 4.3.1), `DelRelFromObjRequest` (described in section 4.3.2), `AddObj2RelRequest` (described in section 4.3.3), `DelObjFromRelRequest` (described in section 4.3.4), `BindApplicationRequest` (described in section 4.4.2), `UnbindApplicationRequest` (described in section 4.4.3), `UnbindUserRequest` (described in section 4.4.4), `ObjectPresentRequest` (described in section 4.4.5), `QueryRequest` and `QueryAbandonRequest` (described in section 4.4.6), `ModifyRequest` (described in section 4.4.7), `JoinGroupRequest` (described in section 4.4.8), `JoinSessionRequest` (described in section 4.4.9), `LeaveSessionRequest` (described in section 4.4.10), `LeaveGroupRequest` (described in section 4.4.11), `DeleteRequest` (described in section 4.4.12), and `RenegotiateRequest` (described in section 4.4.13).

The unicast infrastructure is used in all cases where the receiver of a PDU is already known. This is done to avoid unnecessary network traffic. Because we use the unicast connection only for sending a single PDU, a transactional variant of TCP such as T/TCP described by Braden [3, 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. PDUs sent by unicast are `JoinDomainResponse` (described in section 4.1.1), `DomainInfo` (described in section 4.1.2), `LeaveDomainResponse` (described in section 4.1.3), `DomainResolveResponse` (described in section 4.1.4), `IdentifyTHResponse` (described in section 4.2.2), `AddRel2ObjResponse` (described in section 4.3.1), `DelRelFromObjResponse` (described in section 4.3.2), `AddObj2RelResponse` (described in section 4.3.3), `DelObjFromRelResponse` (described in section 4.3.4), `BindUserIndication` (described in section 4.4.1), `BindApplicationResponse` (described in section 4.4.2), `UnbindApplicationResponse` (described in section 4.4.3), `UnbindUserResponse` (described in section 4.4.4), `CreateRequest` and `ObjectPresentResponse` and `CreateResponse` (described in section 4.4.5), `QueryResponse` (described in section 4.4.6), `ModifyResponse` (described in section 4.4.7), `JoinGroupResponse` (described in section 4.4.8), `JoinSessionResponse` and `AdmissionControlRequest` and `AdmissionControlResponse` (described in section 4.4.9), `LeaveSessionResponse` (described in section 4.4.10), `LeaveGroupResponse` (described in section 4.4.11), `DeleteResponse` (described in section 4.4.12), `RenegotiateResponse` (described in section 4.4.13), `ManagerRequest` and `ManagerResponse` (described in section 4.4.14), `NotificationRequest` (described in section 4.4.15), `InvitationRequest` (described in section 4.4.16), and `RenegotiationRequest` (described in section 4.4.17).

### 3.3 Domain information

Information about domain members is periodically exchanged between all GSAs of a domain. This information is simply sent to the domain's multicast address. The PDU used for this purpose is described in section 4.1.2. Basically, a table consisting of status information about GSAs is transmitted. This table is called a GSA table. Table 1 shows an example for such a table.

| unicast address | member | version |
|---|---|---|
| 129.132.66.9 | yes | 6 |
| 129.132.66.6 | no | 3 |
| 192.35.149.197 | yes | 1 |

Table 1: GSA table (example)

The first column contains the unicast address of a GSA, in the example this is an IP address. The second column contains the member information, which specifies whether the GSA currently is a member of the domain or not (it can have the values "yes" and "no"). This status is changed if a join or leave domain request from this GSA is received (the exact procedure is described in section 3.4). The last column is a version count and helps GSAs to keep their GSA tables up to date.

If a GSA receives a domain information for its domain, it compares its own table with the table it received. If the received table contains newer entries than the own table (which can be detected using the version field) or lines for GSAs which are not in the own table, it is updated accordingly. This way it can be assured that the most actual information about a GSA is distributed inside a domain. The number of GSAs which are currently member of the domain can be obtained from counting the lines of the GSA table where the member field is set to "yes".

### 3.4 GSA startup and shutdown

Domains are the major structuring mechanism of GSAs. For a number of operations (described in section 3.6) it is necessary for a GSA to know the number of GSAs in the domain. For a GSA it is therefore necessary, to keep track with all GSAs entering and leaving the domain. Another mechanism, described in section 3.3 permits the exchange of information between a domain's GSAs and therefore makes it easier for GSA to keep their information about the domain up to date. However, it is still necessary for GSA to conform to the rules defined in the current section when joining or leaving a domain.

If a GSA wants to join a domain, it first needs to be configured in a way that it knows the domain's multicast address and also the multicast addresses of all neighboring domains. As the first step, the GSA has to join the domain's multicast address. The result of this step is that all messages addressed to the domain now are also delivered to the new GSA. The second step is to send a join domain request to the domain (described in detail in section 4.1.1), which contains only a unique request identification. All GSAs receiving the join domain request will respond with a join domain response (also described in section 4.1.1). After the new GSA received the first join domain response, it has successfully joined the new domain. However, the domain information has to be received first before the new GSA knows about the domain's GSAs.

A GSA receiving a join domain request performs the following steps. If the GSA table already contains an entry for this GSA with the member field set to "yes", the receiving GSA does not change its GSA table and responds with a join domain response. If there is an entry with the member field set to "no", the member field is changed, the version number is incremented, and a join domain response is sent. If there is no entry, a new entry with the member field set to "yes" and a version

number of 1 is generated a a join domain response is sent. In all three cases, a domain information is sent to the new GSA using a unicast connection directly after the join domain response has been sent.

If a GSA wants to leave a domain, it is first necessary to move all objects and relations from this GSA to other GSAs of the domain. Otherwise it would be possible that duplicates are created while the GSA is outside the domain which would introduce errors if the GSA joined the domain again at a later point in time. This moving of objects can be performed using the GSP services delete and create (described in sections 4.4.12 and 4.4.5), but it could also be performed with some mechanism outside GSP, such as merging two databases. After all objects and relations have been moved, the leaving GSA sends a leave domain request to the domain (described in detail in section 4.1.3), which contains only a unique request identification. As soon as the first leave domain is received, the GSA leaves the domain's multicast group and initiates a token negotiation for every token it holds (see section 3.5 for details about GSP tokens).

A GSA receiving a leave domain request performs the following steps. If there is an GSA table entry for this GSA with the member field set to "yes", it is set to "no", the version number is incremented, and a leave domain response is sent. If there is no entry in the GSA table, the leave domain request is ignored.

## 3.5 Tokens

GSP uses a multicast infrastructure (described in section 3.2) which makes it possible to transmit PDUs to all GSAs of a domain. One concept used for GSP to deal with the complexity introduced by multicast communications is the one of tokens. Tokens exist inside a domain and are used to determine which GSA inside a domain in 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 domain name resolution requests.* Because it is often necessary to propagate domain name resolution requests (which are described in detail in section 3.6.1) 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. It is shown in figure 3. 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 (ie an `InitToken` PDU as described in section 4.2.1) 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. Because GSAs must wait for other GSAs to send their claim token messages, the final decision is delayed until a timeout, which is identical for all GSAs of a domain. Only after this timeout the GSAs change their state according to the result of the token negotiation.

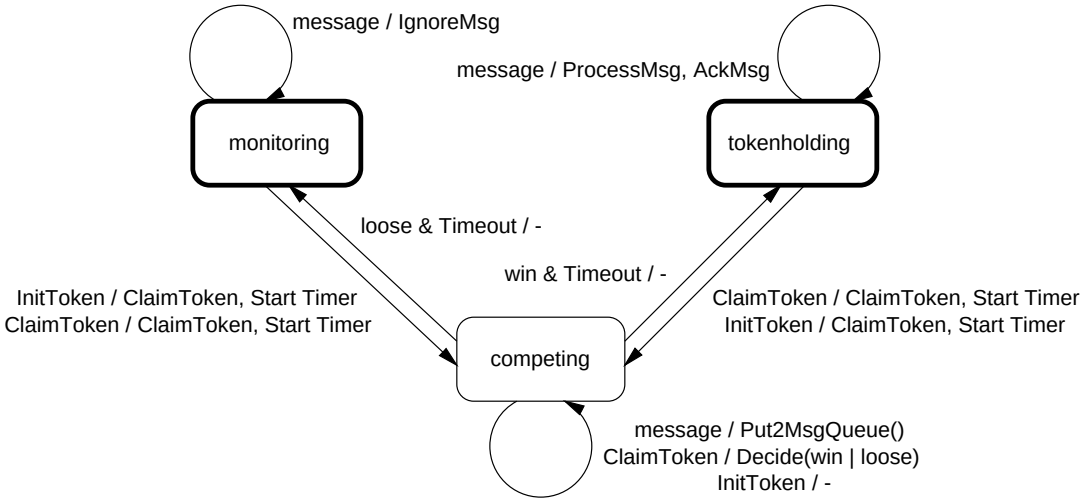


Figure 3: Token negotiation finite state machine

The GSA winning the token negotiation then enters the tokenholding state, while all other GSAs will become monitoring. The token renegotiation process normally 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. It can also be initiated by a GSA holding a token and leaving a domain.

The token negotiation itself is based on the contents of the claim token PDUs (described in detail in section 4.2.1). Each claim token PDU contains a priority, a load, and an address. The GSA with the highest priority wins the token. If there is more than one GSA with the highest priority, the GSA with the smallest load wins the token. If there are more than one GSA with the highest priority and the smallest load, the GSA with the smallest address wins the token. This decision is always possible, since the address of a GSA is unique.

During token negotiation (ie when being in the competing state), messages which need to be processed by a token holder are buffered in a queue and their processing is delayed until the new token holder is evaluated. Each GSA which does not get the token, discards all messages from this queue after entering the monitoring state, while the GSA which got the token enters the tokenholding state and processes all messages it buffered. Using this strategy, it is possible to reduce the number of retransmissions which are needed for GSAs requesting services which need to be processed by token holders.

## 3.6 GAP initiated operations

Whenever a user of the GMS issues a GAP request, the GSA which receives this request will perform a number of operations which depend on the type of request it received. The basic process of GAP initiated operations being carried out within GSP can be separated into two phases. The first phase is the domain name resolution, which is described in section 3.6.1. This phase is only necessary if the GSA does not know the domain's address. The second phase is the operation itself, which can be sent to the domain after its address has been evaluated. This phase is described in section 3.6.2.

However, if a GSA caches domain addresses, it may not always be necessary to perform a domain name resolution, because already resolved names can be resolved using the cache.

### 3.6.1 Domain name resolution

As mentioned in section 3.1, GMS 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. 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 section 3.5). This process, which is shown in figure 4, continues until the address of the requested domain is found, which is then directly sent back to the initiating GSA.

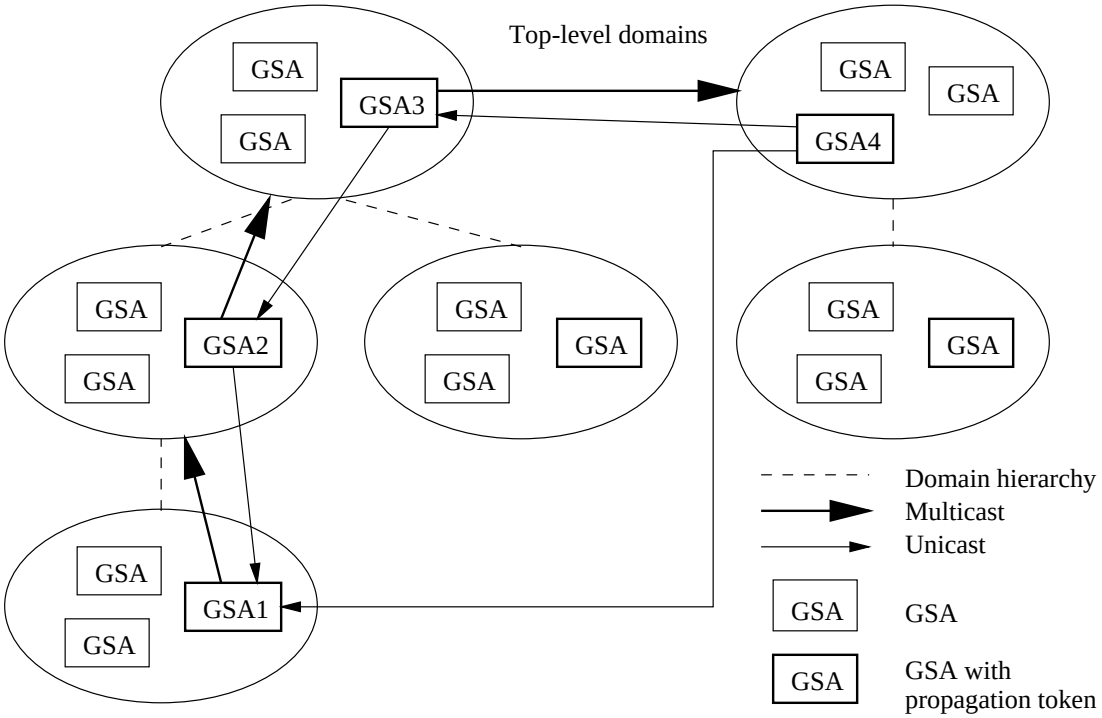


Figure 4: GSP domain name resolution

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.

### 3.6.2 GAP operation processing

The second step of processing a GAP initiated operation is the processing of the operation itself. This step takes place after the domain name resolution discussed in the previous section. Therefore, for this second step the multicast address of the domain to which the PDU has to be sent in already known. The remainder of the section will explain the various ways in which GAP initiated operations are performed.

A GAP bind user operation can be processed locally by a GSA (by getting the user object and performing the authentication and authorization locally), but after successful binding, the GSA storing the user object must be informed about the binding and update the user object accordingly. This is done by using the PDU described in section 4.4.1, which is sent using multicast. Depending on the notification policy of the groups the user is a member of, the GSA storing the user object sends notifications to group members and/or managers.

The bind and unbind application services of GAP are simply mapped onto similar GSP PDUs (described in sections 4.4.2 and 4.4.3) which are then sent to the domain where the user object is stored. The reply of the GSA storing the user object is converted from a GSP PDU to a GAP PDU and forwarded to the user who requested the bind or unbind application service.

A user requesting an unbind from GMS using the GAP unbind user service will cause GSP unbind user service PDUs (described in section 4.4.4) to be sent. The GSA storing the user object will decide whether the unbind can be performed or not and send the appropriate response, which is then forwarded to the user using the according GAP PDU. Depending on the notification policy of the groups the user is a member of, the GSA storing the user object sends notifications to group members and/or managers

The GAP create service requires a more complex procedure, because it must be guaranteed that no duplicates are created (the PDUs required for the create service are described in section 4.4.5). To ensure this, it is first checked whether there is an object with the requested name inside the domain and whether the GSAs attempting the object creation is the only create token holder inside the domain. Only after this check made sure that the requested name is unused and no other GSA will be able create an object of the same name, the new object or relation will be created. Figure 5 shows the exact procedure, with lighter arrows indicating unicast transmissions and bold arrows indicating multicast transmissions.

In the figure, the successful create procedure is depicted, ie the object to be created does not exist, no other token holders were discovered, and thus the object can be created by the create token holder. After a GSA has received a GAP create request, it performs a domain name resolution for the domain of the object to be created. It then sends an identify token holder request for the create token (described in section 4.2.2) to the domain. The GSA holding the create token responds and then gets the create request using a unicast connection (this eliminates problems which could be caused by multiple create token holders inside a domain).

The create token holding GSA then has to check whether an object of the requested name already exists. It does so by sending an object present request (described in section 4.4.5) to its own domain. Only if all GSA of the domain reply with a not found response, the object can be created. In most cases, it is also necessary to establish relations with other objects, which is done using the add relation to object service described in section 4.3.1. If all these steps have been completed, a create response can be sent to the originating GSA which then sends a GAP create response to the user's GUA.

The GAP query service can be used in three variants, using a local mode, a domain list mode, or a global mode. In all three cases, the GSP query service (described in section 4.4.6) is used to perform the query. A local query is simply processed by sending a query request to the domain's multicast address, and each GSA replies with a query response. The originating GSA then collects the responses and eventually sends a GAP query response to the user's GUA.

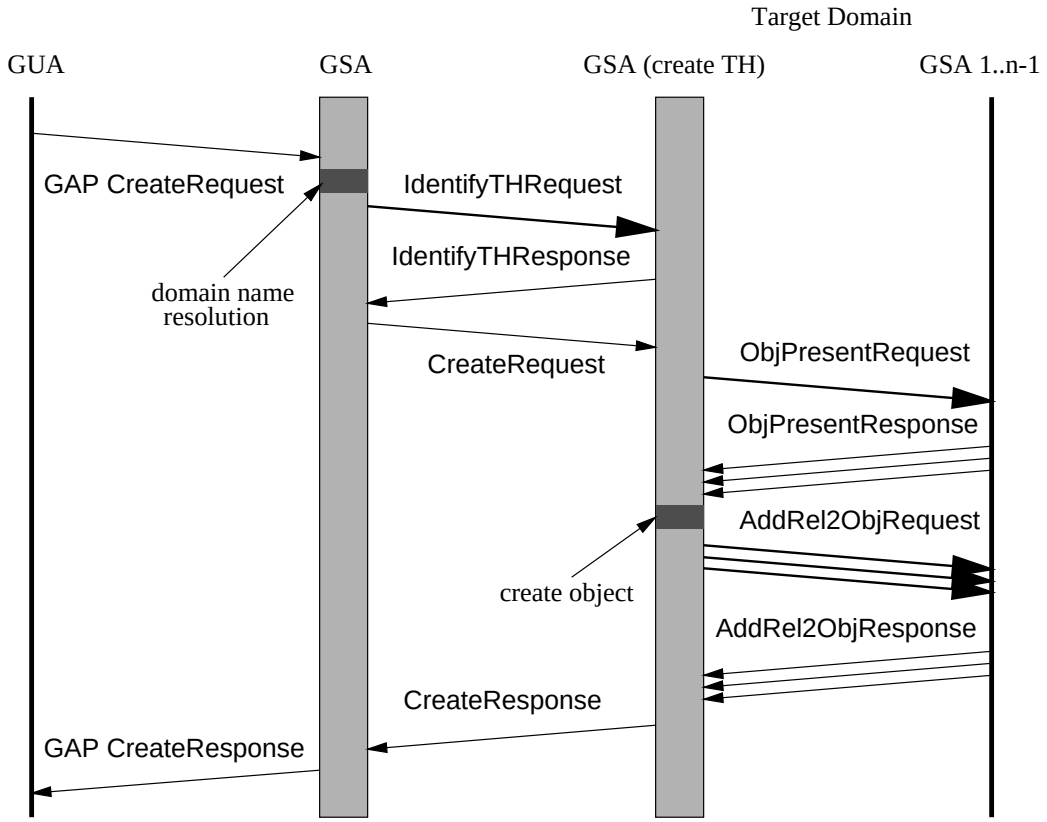


Figure 5: Time sequence diagram for GSP create service

A domain list query is processed by first performing a domain name resolution for all requested domains. The originating GSA then sends query requests to all these domains using their multicast addresses. In each domain, the query token holder responds immediately with a query response with a result of query processing, which is sent using unicast to the originating GSA. The query token holder then performs a local query request, collects all replies and eventually sends a query response containing the results to the originating GSA. The originating GSA collects all results from the domains in the domain list and finally sends a GAP query response to the user's GUA.

A global query requires the most complex procedure. In GSP, it is designed to recursively perform queries in domains, where each query token holder propagates the request to all other neighboring domains (except the one from which it received the query request). In particular, if a GSA receives a query request with a global query and it is the query token holder, it immediately replies with a query response with a result of query processing, which is sent using unicast to the GSA from which it received the query request. It then sends global query requests to all subdomains and the superdomain (or all other top-level domains, if it is within a top-level-domain), except the domain from which it received the query request. Finally, the query token holder performs a local query request. It then collects all replies (from the local query and from the queries it sent to the neighboring domains) and eventually sends a query response containing the results to the GSA from which it received the query request.

All three types of queries can be abandoned by the user by sending a GAP query abandon request. The dissemination of query abandon requests is identical to this of query requests. This way, query abandon requests are distributed in the same way as query requests. Depending on the query abandon request's abandon type (as described in section 4.4.6, the GSAs receiving a query abandon request either response with partial results or with no results at all. In any case, the

distribution of query requests is stopped, ie no GSA receiving a query abandon request will send any more query requests.

The modify service (described in section 4.4.7), the join group service (described in section 4.4.8), the leave session service (described in section 4.4.10), the leave group service (described in section 4.4.11), the delete service (described in section 4.4.12), and the renegotiate service (described in section 4.4.13) are all processed in the same way. After receiving a GAP request, the receiving GSA performs a domain name resolution and then sends the request to the domain where the object for which the service has been requested is stored. The GSA storing the object performs the necessary processing (which usually requires local operations as well as other GSP services, eg for creating or removing relations between objects) and finally sends back a response using a unicast connection. The originating GSA then sends an appropriate GAP response to the GUA. In case of the join group service, depending on the group's notification policy, the GSA storing the group object sends notifications to group members and/or managers. In case of the create and delete services, if they are used to create or delete sessions associated to a group and this group's notification policy is set accordingly, the GSA storing the session object sends notifications to group members and/or managers.

Joining a session using the GAP join session service requires a little more interactions. The PDUs for this service are defined and explained in section 4.4.9. After receiving a GAP join session request, the GSA performs a domain name resolution and sends a join session request to the domain where the session is stored. After performing an authorization check, the GSA storing the session object replies with a join session response. If the authorization check was successful and the session join requires manager requests, the GSA send manager requests to all managers of the session which are bound to the GMS. If the manager requests were successful or no manager requests were needed, the GSA sends an admission control request to the originating GSA using a unicast connection. The GSA forwards the admission control request to the GUA using a GAP admission control request and waits for an GAP admission control response. After receiving this response, it is forwarded to the GSA storing the session object using an admission control response and a unicast connection. The GSA storing the session object then decides whether the session join has been successful or not and replies with an appropriate join session response using a unicast connection. The originating GSA forwards this as a GAP join session response to the GUA. Depending on the session's notification policy, the GSA storing the session object sends notifications to session participants and/or managers.

Whenever join group or join session requests are used for non-open groups or sessions, the managers of the group or session must be requested. This is done by the GSA storing the group or session object using the manager service (described in section 4.4.14). The manager request is sent to the GSA to which the user who shall receive the manager request is bound using a unicast connection. This GSA then creates an GAP manager request PDU and sends it to the user's GUA. After the user replied with a GAP manager response PDU, the GSA creates a according manager response PDU and sends it to the GSA storing the group or session object (ie the GSA which initiated the manager request).

The notification service (described in section 4.4.15), the invitation service (described in section 4.4.16), and the renegotiation service (described in section 4.4.17) are all processed in the same way. All these services are initiated by GSAs (in fact, they are indirectly initiated by GUAs performing GAP operations which require notification, invitation, or renegotiation requests to be sent). The request is sent to the GSA to which the user who shall receive the request is bound using a unicast connection. This GSA then creates an according GAP PDU and sends it to the user's GUA.

# 4 GSP PDU Definitions

The PDUs exchanged by the communicating entities (two GSAs) are encoded as follows. The first two bytes are a unsigned coded length field in network byte order specifying the length of the following data. The following data is a PDU as defined in this section, coded according to the basic encoding rules (BER) as specified by the ITU [11].

The following module heading for the definition of the GSP PDUs first lists all the necessary imports from the modules described earlier in this specification and then defines some identification types to be octet strings. The definitions of `GspPdu` are necessary for decoding incoming PDUs properly. They are simply a list of all PDUs making up GSP.

```
GSP-PDUS DEFINITIONS ::= BEGIN

IMPORTS
  Application, AuthType, AuthLevel, GmsDomainName, GmsObjectName,
  GmsRelationName, GuaAddress, TransportService
        FROM GMS-COMMON

  User, UserAttributes, Group, GroupAttributes, Flow,
  FlowTemplateAttributes, FlowAttributes, Session, SessionAttributes,
  CertificateAttributes, GmsObject, GmsObjectAttributes
        FROM GMS-OBJECT-TYPES

  QosParameter, QosRenegotiation
        FROM GMS-QOS

  Dependency, Synchronization, GmsRelation
        FROM GMS-RELATIONS

MulticastAddress ::= OCTET STRING

UnicastAddress ::= OCTET STRING

UniqueIdentifier ::= SEQUENCE {
  requestID                     [0] INTEGER,
  GSAaddress                    [1] UnicastAddress }

UserGroups ::= -- snacc isPdu:"True" -- SET OF GmsObjectName

GspPdu ::= --snacc isPdu:"TRUE" -- CHOICE {
  domainResolveRequest          [0]     DomainResolveRequest,
  domainResolveResponse         [1]     DomainResolveResponse,
  initToken                     [2]     InitToken,
  claimToken                    [3]     ClaimToken,
  addRel2ObjRequest             [4]     AddRel2ObjRequest
  addRel2ObjResponse            [5]     AddRel2ObjResponse,
  delRelFromObjRequest          [6]     DelRelFromObjRequest,
  delRelFromObjResponse         [7]     DelRelFromObjResponse,
  addObj2RelRequest             [8]     AddObj2RelRequest,
  addObj2RelResponse            [9]     AddObj2RelResponse,
```

```
delObjFromRelRequest          [10]    DelObjFromRelReuestq,
delObjFromRelResponse         [11]    DelObjFromRelResponse,
joinDomainRequest             [12]    JoinDomainRequest,
joinDomainResponse            [13]    JoinDomainResponse,
domainInfo                    [14]    DomainInfo,
leaveDomainRequest            [15]    LeaveDomainRequest,
leaveDomainResponse           [16]    LeaveDomainResponse,
objectPresentRequest          [17]    ObjectPresentRequest,
objectPresentResponse         [18]    ObjectPresentResponse,
createRequest                 [19]    CreateRequest,
createResponse                [20]    CreateResponse,
queryRequest                  [21]    QueryRequest,
queryResponse                 [22]    QueryResponse,
queryAbandonRequest           [23]    QueryAbandonRequest,
modifyRequest                 [24]    ModifyRequest,
modifyResponse                [25]    ModifyResponse,
joinGroupRequest              [26]    JoinGroupRequest,
joinGroupResponse             [27]    JoinGroupResponse,
leaveGroupRequest             [28]    LeaveGroupRequest,
leaveGroupResponse            [29]    LeaveGroupResponse,
joinSessionRequest            [30]    JoinSessionRequest,
joinSessionResponse           [31]    JoinSessionResponse,
admissionControlRequest       [32]    AdmissionControlRequest,
admissionControlResponse      [33]    AdmissionControlResponse,
leaveSessionRequest           [34]    LeaveSessionRequest,
leaveSessionResponse          [35]    LeaveSessionResponse,
deleteRequest                 [36]    DeleteRequest,
deleteResponse                [37]    DeleteResponse,
renegotiateRequest            [38]    RenegotiateRequest,
renegotiateResponse           [39]    RenegotiateResponse,
renegotiationRequest          [40]    RenegotiationRequest,
managerRequest                [41]    ManagerRequest,
managerResponse               [42]    ManagerResponse,
notificationRequest           [43]    NotificationRequest,
invitationRequest             [44]    InvitationRequest,
identifyTHRequest             [45]    IdentifyTHRequest,
identifyTHResponse            [46]    IdentifyTHResponse,
bindUserIndication            [47]    BindUserIndication,
unbindUserRequest             [48]    UnbindUserRequest,
unbindUserResponse            [49]    UnbindUserResponse,
bindApplicationRequest        [50]    BindApplicationRequest,
bindApplicationResponse       [51]    BindApplicationResponse,
unbindApplicationRequest      [52]    UnbindApplicationRequest,
unbindApplicationResponse     [53]    UnbindApplicationResponse }
```

GSP PDUs can be grouped into different categories, depending on the context where they are used. The categories used in this section are mainly derived from the GSP procedures described in section 3. However, since some PDUs are used in different contexts, a slightly different structure has been chosen.

Section 4.1 describes all PDUs which are needed for domain management within GSP. Specifi-

cally, these are PDUs for joining and leaving domains (as described in section 3.4), for periodically transmitting the domain information of each GSA to all other GSAs of a domain (as described in section 3.3), and for domain name resolution (as described in section 3.6.1).

In section 4.2, all PDUs regarding the handling of tokens (as described in section 3.5) are described. These are PDUs for token negotiation and PDUs for token holder identification.

Since almost all objects in GMS are linked through relations, it is very often necessary to modify relations. Section 4.3 therefore describes PDUs for relation management, including PDUs to add and delete relation to respectively from an object and to add and delete an object to respectively from a relation.

The biggest number of PDUs is dealing with GAP initiated operations as described in section 3.6. These PDUs are described in section 4.4. Most GAP PDUs have corresponding GSP PDUs, however there are always small differences between both protocols.

## 4.1 Domain management

Domain management is used in different contexts of GSP. The first context is the startup and shutdown of GSAs (described in section 3.4), which requires the PDUs for joining a domain (described in section 4.1.1) and leaving a domain (described in section 4.1.3). The second context is the periodic exchange of domain information (described in section 3.3), which requires the PDU for transmitting the domain information (described in section 4.1.2). The third context is the performing of GAP initiated operations (described in section 3.6), which requires the PDUs for domain name resolution (described in section 4.1.4).

### 4.1.1 Join domain

When a GSA starts up, it has to join a domain. It does so by announcing its presence to all other GSAs of the domain. The other GSAs then respond to the GSA. The procedure of GSA startup is described in detail in section 3.4.

```
JoinDomainRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier }
```

The `JoinDomainRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. All GSAs receiving a `JoinDomainRequest` PDU update their GSA tables accordingly (as described in section 3.4).

```
JoinDomainResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier }
```

The `JoinDomainResponse` PDU is sent from all GSAs which received the `JoinDomainRequest` PDU to the requesting GSA using a unicast connection. The requesting GSA considers the `JoinDomainRequest` to be successful after the first `JoinDomainResponse` received.

### 4.1.2 Domain information

To ensure that each GSA of a domain knows about all other GSAs of a domain, GSP uses the concept of GSA tables. GSA tables and their usage are described in sections 3.4 and 3.3. Normally, they are only modified when a GSA joins (described in section 4.1.1) or leaves (described in section 4.1.3) the domain. However, because a PDU may be lost due to network failure and it is essential that the GSA tables are correct in all GSAs of a domain, the tables are periodically exchanged inside

a domain. Whenever a GSA receives this information, it compares it with its own GSA table and updates it, if necessary.

```
DomainInfo ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  table         [0]     SET OF SET {
    unicastAddress      [0]     UnicastAddress,
    upDown              [1]     BOOLEAN,
    version             [2]     INTEGER } }
```

The `DomainInfo` PDU is sent to a domain using the domain's multicast address. It is also sent to the new GSA using a unicast connection directly after a GSA has joined a domain. Each GSA sends its GSA table periodically to the domain. The `table` attribute of the `DomainInfo` PDU contains the sending GSA's GSA table. Each entry consists of a GSA's `unicastAddress`, an `upDown` flag indicating whether the GSA is currently running or not, and a `version` number which is used to determine the most actual information concerning a GSA. A detailed description of the GSA table and its usage is given in section 3.3.

### 4.1.3 Leave domain

When a GSA shuts down, it has to leave the domain. It does so by announcing its disappearance to all other GSAs of the domain. The other GSAs then respond to the GSA. The procedure of GSA shutdown is described in detail in section 3.4.

```
LeaveDomainRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  leavingGsa    [1]     UnicastAddress OPTIONAL }
```

The `LeaveDomainRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. All GSAs receiving a `LeaveDomainRequest` PDU update their GSA tables accordingly (as described in section 3.4). The optional `leavingGsa` attribute is used if a running GSA wants to perform the domain leave for another GSA. In this case, the `leavingGsa` attribute is used for updating the GSA tables instead of the `GSAaddress` from the `requestID`'s `UniqueIdentifier`. This is useful if a GSA crashed and is not able to leave the domain properly by itself.

```
LeaveDomainResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier }
```

The `LeaveDomainResponse` PDU is sent from all GSAs which received the `LeaveDomainRequest` PDU to the requesting GSA using a unicast connection. The requesting GSA considers the `LeaveDomainRequest` to be successful after the first `LeaveDomainResponse` received. It may then shut down and therefore ignore other `LeaveDomainResponse` PDUs which are sent by other GSAs.

### 4.1.4 Domain name resolution

GSP operations are performed in two stages, the first one being the domain name resolution, the second one being the operation itself. This separation into two phases is described in section 3.6. The operations which may be used after successful domain name resolution are described in section 4.4. The PDUs described in this section are used for domain name resolution, which is described in detail in section 3.6.1.

```
DomainResolveRequest::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  domain        [1]     GmsDomainName }
```

A `DomainResolveRequest` PDU is sent to a domain using the domain's multicast address. This address is either known because of the GSA's configuration or because of an earlier domain name resolution request. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `domain` attribute specifies the domain name for which the address is required.

```
DomainResolveResponse::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  result        [1]     ENUMERATED {
    success                 (1),
    nonExistingDomain       (2),
    resolvePending          (3) },
  domain        [2]     MulticastAddress OPTIONAL }
```

The `DomainResolveResponse` PDU is sent from a token holder to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `result` attribute indicates the outcome of the domain name resolution. If it is set to `success`, then the optional `domain` attribute contains the multicast address of the domain. A `DomainResolveResponse` with a `result` of `success` is only possible if the GSA processing the `DomainResolveRequest` knew the domain's address through its configuration. Normally, such a GSA sends two `DomainResolveResponse` PDUs, one to the GSA from which it got the `DomainResolveRequest` with a result of `resolvePending`, indicating that it received the `DomainResolveRequest` PDU, and one to the originator of the domain name resolution request, ie the GSA which initially started the domain name resolution (the exception to this rule is the case of a GSA which receives the `DomainResolveRequest` PDU directly from the originator and already knows the domain's address).

The `result` attribute of the `DomainResolveResponse` PDU is set to `nonExistingDomain` if the GSA processing the `DomainResolveRequest` PDU determines that the domain searched for was an immediate sub-domain or super-domain of its own domain but has no entry for it in its configuration. In this case it can be safely concluded that the domain does not exist. Like in the first case (a result of `success`), it is also the normal case that two `DomainResolveResponse` PDUs have to be sent, one with a result of `resolvePending` to the GSA from which the `DomainResolveRequest` was received, and one with a result of `nonExistingDomain` to the originator of the domain name resolution request.

If the GSA processing the `DomainResolveRequest` PDU can not determine whether the requested domain does exist or not (because it is not an immediate sub-domain or super-domain), it sends a `DomainResolveResponse` PDU with a `result` of `resolvePending` to the GSA which sent the `DomainResolveResponse` PDU and forwards the `DomainResolveResponse` PDU. Forwarding the `DomainResolveResponse` PDU is simply done by sending the PDU to the domain which is nearer to the searched domain than the own domain. This can easily be concluded from the own domain name and the `domain` given in the `DomainResolveRequest` PDU.

## 4.2 Token management

Because many messages in GSP are sent with multicast, for some operations it is necessary to identify token holders which are the only GSAs inside a domain which may process certain requests. Basically, token management includes the tasks of negotiating a token, ie determining which GSA should get a token (described in section 4.2.1), and token holder identification, ie determining which GSA inside a domain currently is token holder (described in section 4.2.2).

```
TokenType ::= ENUMERATED {
  propagate     (1),
  create        (2),
  query         (3) }
```

There are three `TokenType` values, ie there are three token types which can be used with the token management services. The `propagate` token is used to determine which GSA is responsible for forwarding domain name resolution requests (described in section 4.1.4). The `create` token is used to determine which GSA is responsible for the creation of objects inside a domain (described in section 4.4.5). The `query` token is used to determine which GSA is responsible for the forwarding and processing of query requests (described in section 4.4.6).

### 4.2.1 Token negotiation

If one of the three tokens is missing inside a domain, the `InitToken` PDU can be used to force a token negotiation. A token is supposed to be missing if no token holder has answered a request inside a domain for a given time. A missing token can be detected after a `DomainResolveRequest` (described in section 4.1.4), after a `IdentifyTHRequest` (described in section 4.2.2) which is used in the context of a create operation (described in section 4.4.5), or after a `QueryRequest` (described in section 4.4.6).

Another trigger for initiating a token negotiation is the detection of a duplicated token, ie the existence of more than one token holder inside a domain. This can be detected by a GSA if it receives more than one response to a request which should be answered only by the token holder. Token duplication can occur due to network partitioning. The reaction to token duplication is the same as the reaction to a missing token, ie a new token holder has to be negotiated.

```
InitToken ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  tokenType     [0]     TokenType }
```

The `InitToken` PDU is sent to a domain using the domain's multicast address. The only parameter of the PDU is the `tokenType` which determines which token is to be negotiated. This PDU can be sent by any GSA requesting a token negotiation inside a domain.

```
ClaimToken ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  tokenType     [0]     TokenType,
  priority      [1]     INTEGER (0..100),
  load          [2]     INTEGER (0..100),
  address       [3]     UnicastAddress }
```

The `ClaimToken` PDU is sent by a GSA which tries to get a token to the domain using the domain's multicast address. The `tokenType` attribute determines which type of token is claimed by the sending GSA. The `priority` attribute specifies the priority which is used for claiming the token. It is an integer value which may be in the range between 0 and 100. A GSA's priority is static and is a configuration parameter of a GSA (ie it is determined at GSA startup).

The `load` attribute determines the dynamic priority of a GSA. Depending on the value of the `tokenType` attribute, the `load` attribute has different semantics. However, in all cases it is an integer value which may be in the range between 0 and 100. If the `tokenType` is `create`, the `load` attribute specifies the disk usage of the GSA. If the `tokenType` is `propagate` or `query`, the `load` attribute specifies the CPU load of the GSA.

Because it is possible that two GSAs in the same domain have the same `priority` and `load` when competing for a token, a third value is used inside the `ClaimToken` PDU. The `address` attribute is

an integer value which uniquely identifies each GSA in a domain. If more than one GSA have the same `priority` and `load` when competing for a token, the GSA with the smallest `address` wins and becomes the token holder.

### 4.2.2 Identify token holder

For some operations it is necessary to identify the token holder inside a domain. In the present version of GSP, this is only used by the create operation described in section 4.4.5. When creating an object it is necessary to be sure that exactly one create token holder exists inside the domain where the object or relation is to be created. To ensure this, a request is sent to the domain which has to be answered by all token holders.

```
IdentifyTHRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  tokenType     [1]     TokenType }
```

The `IdentifyTHRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request, and a `tokenType` attribute which determines the token type for which this identification request is being sent. At present, this attribute always is set to `create`, because this is the only token type for which an identification request is used.

```
IdentifyTHResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier }
```

The `IdentifyTHResponse` PDU is sent from a token holder to the requesting GSA using a unicast connection. All GSAs in a domain which consider themselves token holders for the requested token have to reply using the `IdentifyTHResponse` PDU. The `requestID` attribute uniquely identifies the request for which this response is being sent. The token type for which this response is given can be concluded from the `requestID` attribute.

## 4.3 Relation management

In GMS, data is being stored in objects (described in section 2.3) and relations (described in section 2.4). Because these may be distributed over different GSAs (maybe in different domains), it is necessary to have operations for modifying them from another GSA. The possible relation types are defined as follows and used in the relation management PDUs.

```
RelationType ::= ENUMERATED {
  association       (1),
  dependency        (2),
  manager           (3),
  member            (4),
  owner             (5),
  part              (6),
  participation     (7),
  receiver          (8),
  sender            (9),
  synchronization   (10) }
```

There are four possible cases how remote relations may be modified. It may be necessary to add a relation to remote object, this service is described in section 4.3.1. The reverse of this operation is the removal of a relation from a remote object. This is done with the delete relation from object service described in section 4.3.2. The two other services are used for adding an object to or deleting an object from a remote relation. They are described in sections 4.3.3 and 4.3.4.

### 4.3.1 Add relation to object

If new relations between objects need to be established, it is necessary to create a new relation (if the relation does not exist already) and to add the required entries to the objects and the relation. The add relation to object service is used to add a relation to a remote object. The normal procedure to establish a relation between a remote and a local object is to create a relation locally (if necessary), to add this relation to the remote object, and to finally add the relation to the local object. The modifications of the local object and relation can be done without GSP operations, because it only requires the local database.

```
AddRel2ObjRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  objectName    [1]     GmsObjectName,
  type          [2]     RelationType,
  relation      [3]     GmsRelationName }
```

The `AddRel2ObjRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `objectName` attribute contains the name of the object to which the relation should be added. The `type` attribute identifies the type of the relation to be added, and the `relation` attribute specifies the name of the relation to be added to the remote object.

```
AddRel2ObjResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  result        [1]     ENUMERATED {
   success                  (1),
   existingRelation         (2),
   otherError               (3) } }
```

The `AddRel2ObjResponse` PDU is sent from the GSA storing the object which was named in the `AddRel2ObjRequest`'s `objectName` to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. If the `result` is set to `success`, the relation has been successfully added to the object. A `result` of `existingRelation` indicates that the relation entry already exists in the object. Any other error, indicated by a `result` of `otherError`, means that the modification of the remote object has not been made.

### 4.3.2 Delete relation from object

If existing relations between objects need to be removed, it is necessary to remove the entries from the objects and the relation, and it may also be necessary to remove the relation. The delete relation from object service is used to remove a relation from a remote object.

```
DelRelFromObjRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
```

```
  objectName     [1]     GmsObjectName,
  type           [2]     RelationType,
  relation       [3]     GmsRelationName }
```

The `DelRelFromObjRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `objectName` attribute contains the name of the object from which the relation should be deleted. The `type` attribute identifies the type of the relation to be deleted, and the `relation` attribute specifies the name of the relation to be deleted from the remote object.

```
DelRelFromObjResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID      [0]     UniqueIdentifier,
  result         [1]     ENUMERATED {
    success                  (1),
    nonExistingRelation      (2),
    otherError               (3) } }
```

The `DelRelFromObjResponse` PDU is sent from the GSA storing the object which was named in the `DelRelFromObjRequest`'s `objectName` to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. If the `result` is set to `success`, the relation has been successfully deleted from the object. A `result` of `nonExistingRelation` indicates that the relation entry does not exist in the object. Any other error, indicated by a `result` of `otherError`, means that the modification of the remote object has not been made.

### 4.3.3 Add object to relation

If new relations between objects need to be established, it is necessary to create a new relation and to add the required entries to the objects and the relation. The add object to relation service is used to add an object to a remote relation. This is necessary if the relation already exists (otherwise the procedure described in section 4.3.1 should be used).

```
AddObj2RelRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID      [0]     UniqueIdentifier,
  relationName   [1]     GmsRelationName,
  object         [3]     GmsObjectName }
```

The `AddObj2RelRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `relationName` attribute contains the name of the relation to which the object should be added. The `object` attribute specifies the name of the object to be added to the remote relation.

```
AddObj2RelResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID      [0]     UniqueIdentifier,
  result         [1]     ENUMERATED {
   success                  (1),
   otherError               (3) } }
```

The `AddObj2RelResponse` PDU is sent from the GSA storing the relation which was named in the `AddObj2RelRequest`'s `relationName` to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. If the `result` is set to `success`, the object has been successfully added to the relation. Any error, indicated by a `result` of `otherError`, means that the modification of the remote relation has not been made.

### 4.3.4 Delete object from relation

If relations between objects need to be removed, it is necessary to remove the entries from the objects and the relation. The delete object from relation service is used to remove an object from a remote relation.

```
DelObjFromRelRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  relationName  [1]     GmsRelationName,
  object        [3]     GmsObjectName }
```

The `DelObjFromRelRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `relationName` attribute contains the name of the relation from which the object should be deleted. The `object` attribute specifies the name of the object to be deleted from the remote relation.

```
DelObjFromRelResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  result        [1]     ENUMERATED {
    success                 (1),
    otherError              (3) } }
```

The `DelObjFromRelResponse` PDU is sent from the GSA storing the relation which was named in the `DelObjFromRelRequest`'s `relationName` to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. If the `result` is set to `success`, the object has been successfully deleted from the relation. Any error, indicated by a `result` of `otherError`, means that the modification of the remote relation has not been made.

## 4.4 GAP operations

Most GAP operations (as specified in GAP [21]) have corresponding GSP PDUs. However, GSP handles the operations differently, partly because of the two-step process described in section 3.6 which includes a domain name resolution before the operation itself can be carried out, partly because GSP is using multicast (see section 3.2 for details) and therefore needs special mechanisms dealing with this.

### 4.4.1 Bind User

When a user binds to the GMS using a GUA and GAP, the GSA to which he is binding first gets the user object from the GSA where it is located. Based on this user object, it is possible to perform the authentication locally, ie the GSA storing the user object does not need to do anything. However, if the user successfully bound to the GSA, this must be registered in the user object. The bind user service is used to perform this registration.

```
BindUserIndication ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  userName              [0]     GmsObjectName,
  address               [1]     GuaAddress,
  authentication        [2]     AuthType,
  transportServices     [3]     SET OF TransportService,
  bindingComment        [4]     IA5String OPTIONAL }
```

The `BindUserIndication` PDU is sent from the GSA to which the user bound to the GSA storing the user object using a unicast connection. The `userName` attribute specifies the user who successfully bound to the GMS. The `address` attribute contains the address to which the user has bound. The `authentication` attribute contains the `AuthType` with which the user bound to the GMS. The `transportServices` attribute is a set of `TransportServices`, specifying which transport services the user has available with this binding. Finally, an optional `bindingComment` can be used to store some information about the binding.

The GSA which receives a `BindUserIndication` PDU modifies the user object according to the information the `BindUserIndication` contains.

### 4.4.2 Bind Application

The GAP bind application service is used by a user to notify the GMS that a new application has been started within the context of a user binding. The new application has to be registered inside the user object. The GSP bind application service is used to modify the user object according to the GAP operation.

```
BindApplicationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  user                  [1]     GmsObjectName,
  application           [2]     Application }
```

The `BindApplicationRequest` PDU is sent to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute identifies the user who is binding the new application. The `application` attribute specifies the application which the user is binding.

```
BindApplicationResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  bindApplicationResult [1]     ENUMERATED {
    success                 (1),
    applicationAlreadyBound (2) } }
```

The `BindApplicationResponse` PDU is sent from the GSA storing the user object which was named in the `BindApplicationRequest`'s `user` to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `bindApplicationResult` attribute indicates the result of the bind application request. If it is set to `success`, the application has been successfully bound (ie it has been added to the user object). If the `bindApplicationResult` is set to `applicationAlreadyBound`, the application has already been bound, ie it is already present in the user object.

### 4.4.3 Unbind Application

The GAP unbind application service is used by a user to notify the GMS that an application which has been bound using the bind application service is now unbound. The application therefore has to be deleted from the user object. The GSP unbind application service is used to modify the user object according to the GAP operation.

```
UnbindApplicationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  user                  [1]     GmsObjectName,
  application           [2]     Application }
```

The UnbindApplicationRequest PDU is sent to a domain using the domain's multicast address. The PDU contains a requestID attribute, which is used to uniquely identify this request. The user attribute identifies the user who is unbinding the application. The application attribute specifies the application which the user is unbinding.

```
UnbindApplicationResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  unbindApplicationResult [1]   ENUMERATED {
        success                 (1),
        noSuchApplication       (2) } }
```

The UnbindApplicationResponse PDU is sent from the GSA storing the user object which was named in the UnbindApplicationRequest's user to the requesting GSA using a unicast connection. The requestID attribute uniquely identifies the request for which this response is being sent. The unbindApplicationResult attribute indicates the result of the unbind application request. If it is set to success, the application has been successfully unbound (ie it has been removed from the user object). If the unbindApplicationResult is set to noSuchApplication, the application which had to be unbound was not found in the user object, ie it was not bound.

### 4.4.4 Unbind User

When a user unbinds from the GMS using the GAP unbind user service, the GSA to which he is bound must determine whether the user is allowed to unbind. This is necessary because the user may still be participant in some sessions, in which case he only may unbind if he specifically forces the unbind. The GSP unbind user service is for the communication between the GSA to which the user is bound and the GSA which stores the user object.

```
UnbindUserRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  userName              [1]     GmsObjectName,
  forceUnbind           [2]     BOOLEAN }
```

The UnbindUserRequest PDU is sent from the GSA to which the user is bound to a domain using the domain's multicast address. The PDU contains a requestID attribute, which is used to uniquely identify this request. The userName attribute specifies the user who want to unbind from the GMS. The forceUnbind specifies whether the user wants to unbind even if there are active sessions (ie sessions which the user joined but did not leave prior to his unbind user request).

```
UnbindUserResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  unbindUserResult      [1]     ENUMERATED {
    success                 (1),
    terminatedSessions      (2),
    activeSessions          (3) },
  activeSessions        [2]     SET OF GmsObjectName OPTIONAL }
```

The UnbindUserResponse PDU is sent from the GSA storing the user object which was named in the UnbindUserRequest's userName to the GSA to which the user is bound (ie the GSA which sent the UnbindUserRequest) using a unicast connection. The requestID attribute uniquely identifies the request for which this response is being sent. If the unbindUserResult attribute is set to success, the user has been unbound successfully. If this attribute is set to terminatedSessions,

the user has also been unbound successfully, but there were active sessions which were terminated (this `unbindUserResult` is only possible if the `forceUnbind` attribute in the `UnbindUserRequest` has been set to true).

The `unbindUserResult` is set to `activeSessions` if the user wants to unbind but is still participant in one or more sessions (this `unbindUserResult` is only possible if the `forceUnbind` attribute in the `UnbindUserRequest` has been set to false). In this case, the optional `activeSessions` attribute contains the names of all sessions in which the user is active.

### 4.4.5 Create

The GAP create service is used to create GMS objects. When creating an object, it must be checked whether an object of this name already exists. Because data storage in GMS is distributed, this check has to done before the create request itself can be issued. The object present service therefore is used to check for the existence of an object of a given name. This check is performed by the GSA holding the create token. Therefore, the GSA initiating the create service first has to find the create token holder using the identify token holder service (described in section 4.2.2).

```
CreateRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  user          [1]     GmsObjectName,
  objectName    [2]     GmsObjectName,
  objectType    [3]     CHOICE {
    user            [0]     UserAttributes,
    group           [1]     SET {
      attributes        [0]     GroupAttributes,
      managers          [1]     SET OF GmsObjectName,
      initialMembers    [2]     SET OF GmsObjectName OPTIONAL},
    flowTemplate    [2]     FlowTemplateAttributes,
    session         [3]     SET {
      attributes        [0]     SessionAttributes,
      managers          [1]     SET OF GmsObjectName,
      associatedWithGroup [2]     GmsObjectName OPTIONAL,
      flows             [3]     SET OF SET {
        flowName            [0]     GmsObjectName,
        flowAttributes      [1]     FlowAttributes,
        flowQosParameters   [2]     SET OF QosParameter,
        flowAddressingInfo  [3]     CHOICE {
          receiverOriented      [0]     OCTET STRING,
          senderOriented        [1]     NULL },
        flowRelations       [4]     SET OF CHOICE {
          dependsOn             [0]     GmsObjectName,
          synchronizedWith      [1]     GmsObjectName } } },
    certificate         [4]     CertificateAttributes } }
```

The `CreateRequest` PDU is sent by the GSA requesting the create service to the GSA which is create token holder in the domain where the object shall be created using a unicast connection. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute contains the name of the user who is requesting the create (ie who issued a GAP create service request). The `objectType` attribute defines which type of object shall be created.

If the objectType attribute is of the user type, the user which shall be created must be defined by all UserAttributes as defined in section 2.3.1. The information which is not marked as OPTIONAL in the definition must be supplied. Optional parts may be ignored, they can be added using the modify service. Initially, the newly created user object has none of the relations defined which are present in the user object's UserRelations.

If the objectType attribute is of the group type, the definition of the initial values is more complicated than in case of a user object. At first, all GroupAttributes as defined in section 2.3.2 must be given. The optional parts can be ignored. The second value is a set of managers which must be specified, it defines all users which shall be managers of the new group. The third value is optional, it allows the specification of initialMembers of the group. The relations of the newly created group are defined as follows. The user who created the group is the only owner of it. The set of managers given in the CreateRequest PDU is the set of managers of the group. If the initialMembers has been used, all users and groups present in this set are the members of the group. Initially, there are groups the new group is a member of and no sessions associated with the group.

If the objectType attribute is of the flowTemplate type, a flow template object will be created. All FlowTemplateAttributes as defined in section 2.3.3 must be given. Optional parts can be omitted. After the flow template is created, the user who requested the creation using the create service will be the owner of the flow template object.

If the objectType attribute is of the session type, a session object will be created. Because the creation of a session object always includes the creation of at least one flow object, this case is the most complicated one (actually, it is always created more than one object). At first, all SessionAttributes must be supplied. Optional parts can be omitted. The second value is a set of managers which must be specified, it defines all users which shall be managers of the new session. The associatedWithGroup attribute is used to specify the group to which the session to be created is associated. The user requesting the create must be member of this group. This attribute is optional, but it must be present if the session's sessionJoinPolicy is set to group in the SessionAttributes (otherwise the session's join policy could not be enforced).

The last attribute of the session type defines the flows to be created. Each flow is characterized by a flowName and FlowAttributes as described in section 2.3.4. QoS parameters of each flow are specified in the flowQosParameters, they are defined according to the definition given in section 2.2. The flowAddressingInformation specifies for each flow hot it is addressed. If addressing is receiverOriented, the flow address has to be specified when creating the session (and the flow). If flow addressing is senderOriented, no information must be supplied (the list of receivers will be created when the first user joins this flow). The last attribute, the flowRelations, are used to specify whether this flow shall depend on another flow (using the dependsOn attribute) or whether this flow has to be synchronized with another flow (using the synchronizedWith attribute).

If the objectType attribute is of the certificate type, a certificate object will be created. All CertificateAttributes as defined in section 2.3.6 must be given. Optional parts can be omitted. After the certificate is created, the user who requested the creation using the create service will be the owner of the certificate object.

```
ObjectPresentRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  objectName    [1]     GmsObjectName }
```

The ObjectPresentRequest PDU is sent from the GSA which received the CreateRequest to its own domain using the domain's multicast address. The PDU contains a requestID attribute, which is used to uniquely identify this request. The objectName attribute specifies the name of the object for which this object present request is being sent.

```
ObjectPresentResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  nofGSA        [1]     INTEGER,
  tHolderFlag   [2]     BOOLEAN,
  result        [3]     ENUMERATED {
    found           (1),
    notFound        (2) } }
```

The `ObjectPresentResponse` PDU is sent from every GSA which received the `ObjectPresentRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` and `tHolderFlag` attributes contain status information of the responding GSA. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `tHolderFlag` specifies whether the responding GSA currently holds the create token. If this attribute is set to true in a `ObjectPresentResponse` PDU, it can be concluded that there is more than one create token holder in the domain and the token holder has to be freshly negotiated (using the token negotiation described in section 4.2.1).

The `result` attribute specifies whether the responding GSA stores an object of the `objectName` given in the `ObjectPresentRequest`. If it is `found`, the object existed on the responding GSA and the requesting GSA therefore may not create an object of this name. If the `result` attribute is set to `notFound` in all `ObjectPresentResponse` PDUs, the create token holder can safely create the object specified in the create request.

```
CreateResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  createResult  [1]     ENUMERATED {
    success                 (1),
    noPermission            (2),
    nameInUse               (3),
    nonExistingManager      (4),
    noSuchGroup             (5),
    noSuchUser              (6),
    alreadyIndirectMember   (7),
    cyclicFlowDependencies  (8),
    nofGsaMismatch          (9),
    timeout                 (10),
    multipleTokenHolder     (11) } }
```

The `CreateResponse` PDU is sent from the create token holder which received the `CreateRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `createResult` attribute specifies the result of the create request. A value of `success` indicates that the create service request was processed successfully and the requested object has been created (including modifications of relations if necessary).

The `createResult` of `noPermission` indicates that the user who requested the create service is not authorized sufficiently to perform the operation he requested. This error occurs if an anonymously bound user tries to create an object or if a user tries to create a session which should be associated with a group he is not a member of.

If the `createResult` has one of the values `nameInUse`, `nonExistingManager`, `noSuchGroup`, or `noSuchUser`, the create request contained a name which is not permitted. In case of `nameInUse`, the name which shall be given to the object to be created exists already. In this case, a different name must be chosen. In case of `nonExistingManager`, `noSuchGroup`, and `noSuchUser`, the user referred to a GMS object which does not exist.

A `createResult` of `alreadyIndirectMember` indicates that it is not possible to join a member directly to the group (using the `initialMembers` attribute), because it is already indirect member of the group. This is only possible if groups are used in the `initialMembers` attribute.

The `createResult` of `cyclicFlowDependencies` indicates that the user created a cyclic graph specifying `dependsOn flowRelations` inside the `flows` attribute in case of requesting a `session` create. A cyclic dependency does not make sense and is therefore rejected by the create service. It is up to the user to delete the cyclic dependency (by removing at least one `dependsOn` value) and use the create service again.

A `createResult` of `nofGsaMismatch` indicates that the create token holder got different `nofGSA` values in the `ObjectPresentResponse` PDUs and is therefore not certain about the number of GSAs in the domain. If the `createResult` is `timeout` the operation timed out at the create token holder (probably because of one GSA of the domain not responding to the `ObjectPresentRequest` PDU). A `createResult` of `multipleTokenHolder` indicates that the create token holder discovered another GSA in the domain also holding the create token. In these last three cases, it would not be safe to create the object and therefore it is not created.

### 4.4.6 Query

The GAP query service is used to query the GMS about objects. In general, there are two query modes available. When using the search mode, GMS can be searched for objects which match a given template. Currently, it is not possible to use wild cards or regular expressions, but this functionality will be introduced in the next version of GSP. In this version of GSP, only perfect matches to attribute values are possible. However, if an attribute's value is empty or set to zero, GMS will not try to match this value but only uses the other attribute values for finding matching objects. The result of searching for objects is a list of object names. The search mode can be used in three different scopes, only in the local domain, in a given number of domains, and globally.

When using the get mode of the query service, it is possible to retrieve objects from the GMS (provided the authorization is sufficient). Thus, the normal procedure of using the query service is to first search for objects using the search mode and then to get selected objects using the get mode. A GSA receiving a GAP query request sends a GSP query request to the domain where the query starts.

```
QueryRequest ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  user          [1]     GmsObjectName,
  userAuthLevel [2]     AuthLevel,
  userGroups    [3]     UserGroups,
  queryType     [4]     CHOICE {
    search          [0]     SET {
      queryTemplate     [0]     CHOICE {
        object              [0]     GmsObject,
        relation            [1]     GmsRelation },
      queryScope        [1]     CHOICE {
        localDomain         [0]     NULL,
        domainList          [1]     NULL,
```

```
      global              [2]     GmsDomainName },
    maxNumberOfMatches [2]    INTEGER },
  get             [1]     CHOICE {
    object                [0]     GmsObjectName,
    relation              [1]     GmsRelationName } } }
```

The `QueryRequest` PDU is sent from the GSA which received the GAP query request to a domain using the domain's multicast address. It may also be sent by a query token holding GSA to a domain using the domain's multicast address in order to propagate queries. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute specifies the user who wants to query the GMS. The `userAuthLevel` and `userGroups` attributes contain the user's authentication level and the groups he is member of, which are necessary for authorization checking.

The `queryType` attribute identifies the query mode which is selected. When using the `search` mode, it is possible to search for specific objects or relations. The `queryTemplate` attribute specifies whether the search is performed for an `object` or a `relation`. In case of a search for an `object`, a `GmsObject` as defined in section 2.3 must be specified. When specifying attribute values to be searched for, the `objectName` attribute and all `objectRelations` will not be interpreted. Any values which should be searched for must be supplied as non-empty (or non-zero) values. Empty and zero values will cause an attribute to be excluded from the search. When searching for a `relation`, the `relationName` will not be interpreted.

The `queryScope` attribute determines the scope of the search. It can have one of three values. When the `queryScope` attribute is set to `localDomain`, only the local domain will be searched (ie the query token holder will not forward the query request to other domains). Because the GSA initiating the query sends `QueryRequest` PDUs to all domains when performing a `domainList` search, it is not necessary to specify the domains in the `QueryRequest` PDU (and thus the `domainList` attribute is empty). The `global` attribute contains the domain name from where the search started. The last attribute of the `search` attribute is the `maxNumberOfMatches` value which specifies how many matches should be reported before stopping the search.

If the `queryType` attribute is set to use the `get` mode, the usage of the query service is much simpler. The GSA can select whether it wants to get an `object` or a `relation`. In both cases, the name must be specified. A `get` mode `QueryRequest` PDU is always sent to the domain where the object or relation is stored. A typical usage of the query service would be to first `search` for an object or relation and then to use the results of the search in another `QueryRequest` to get an object or relation.

```
QueryAbandonRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  abandonType   [1]     ENUMERATED {
    dismissResults          (1),
    sendPartialResults      (2) } }
```

Because the query service may take a long time to complete, it is possible for a user to abandon a running query request with the GAP `QueryAbandonRequest` PDU. This PDU may be sent by the GSA at any time after sending a `QueryRequest` PDU and before receiving a `QueryResponse` PDU. It is sent to a domain using the domain's multicast address. The GSP `QueryAbandonRequest` PDU, which is sent by a GSA to all domains to which it sent the `QueryRequest` PDU, has only two attributes, the first one being the `requestID`, which is used to uniquely identify this request. The second attribute is the `abandonType` which may be either set to `dismissResults` or to `sendPartialResults`. In case of `dismissResults`, the query service is abandoned and the GSA will not receive any results in `QueryResponse` PDUs. If the `abandonType` is set to `sendPartialResults`, the

query service will also be abandoned, but the results collected until the abandon will be returned in QueryResponse PDUs.

```
QueryResponse ::= -- snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  domain        [1]     GmsDomainName,
  nofGsa        [2]     INTEGER,
  queryResult   [3]     ENUMERATED {
    success                 (1),
    abandonedNoResults      (2),
    abandonedPartialResults (3),
    permissionDenied        (4),
    tooManyMatchesRequired  (5),
    queryProcessing         (6),
    notFound                (7) },
  queryMatches  [4]     CHOICE {
    search          [0]     CHOICE {
      objects           [0]     SET OF GmsObjectName,
      relations         [1]     SET OF GmsRelationName },
    get             [1]     CHOICE {
      object            [0]     GmsObject,
      relation          [1]     GmsRelation } } OPTIONAL }
```

The QueryResponse PDU is sent from every GSA which received the QueryRequest PDU to the requesting GSA using a unicast connection. Details on how queries are processed can be found in section 3.6. The requestID attribute uniquely identifies the request for which this response is being sent. The domain attribute specifies from which domain this response is being sent. The nofGSA attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The queryResult specifies the result of the query request. A value of success indicates the success of the query service, ie it completed normally without being interrupted by a QueryAbandonRequest. If the query was abandoned, the queryResult is either abandonedNoResults or abandonedPartialResults, depending on the abandonType in the QueryAbandonRequest PDU.

If a user tried to get an object he is not allowed to read, a queryResult of permissionDenied will be returned. The authorization may have failed because the user's authentication level was too low (ie lower than the authRequirements of the object) or because the object's access policy did not give him the read right in the AccessRight as defined in section 2.1. If the tooManyMatchesRequired value is present in the queryResult, more matches than defined in the request's maxNumberOfMatches are necessary to give a complete list of results.

If the QueryRequest's queryScope has been set to domainList or global, the receiving GSA holding the query token replies with a QueryResponse PDU with a queryResult of queryProcessing, indicating that it is processing the query request. If a GSA replies to a query request which used the get mode and does not store the desired object or relation, it responds with a QueryResponse with a queryResult of notFound.

In case of a queryResult of success, abandonedPartialResults, and tooManyMatchesRequired, the QueryResponse PDU contains the optional queryMatches attribute which contains the result of the query service. Depending on the mode of the query service, either a search or a get result is returned. In case of search, the result is a set of names which identify the objects which were found using the specified search criteria. Whether this set is complete or not depends on the

response's `queryResult` value. In case of get, the `queryMatches` attribute contains an `object` or a `relation`, depending on what the user requested.

### 4.4.7 Modify

The GAP modify service is used to modify a GMS object or a GMS relation. Because the internal attributes of an object (as defined in section 2.3) can not be changed by a user (they are modified by the GMS when using certain GAP services), only the object's attributes can be modified. In case of the modification of relations, only two relations may be modified by a user directly, all other relations are only modified by the GMS during internal operations. These two relations are manager and owner, where the modify makes it possible to add new managers or owners to a relation or to remove managers or owners from a relation. A GSA receiving a GAP modify request sends a GSP modify request to the domain where the object is stored.

```
ModifyRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  user          [1]     GmsObjectName,
  userAuthLevel [2]     AuthLevel,
  userGroups    [3]     UserGroups,
  modifyType    [4]     CHOICE {
    object          [0]     SET {
      objectName        [0]     GmsObjectName,
      attributes        [1]     GmsObjectAttributes },
    relation        [1]     SET {
      relationName      [0]     GmsRelationName,
      relationType      [1]     CHOICE {
        manager             [0]     CHOICE {
          addManager            [0]     GmsObjectName,
          deleteManager         [1]     GmsObjectName },
        owner               [1]     CHOICE {
          addOwner              [0]     GmsObjectName,
          deleteOwner           [1]     GmsObjectName } } } } }
```

The `ModifyRequest` PDU is sent from the GSA which received the GAP modify request to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute specifies the user who wants to modify an object. The `userAuthLevel` and `userGroups` attributes contain the user's authentication level and the groups he is member of, which are necessary for authorization checking.

The `modifyType` attribute specifies the modify operation. It may specify an `object` or a `relation` type. In case of an `object` type, the `modifyType` attribute contains the `objectName`, which identifies the object to be modified. The `attributes` attribute specifies the attributes to be changed. Only attributes which are modifiable are interpreted (eg it is not possible to change an object's `objectName`), if an attribute does not have to be changed and it is not optional, its value must be set to the existing value (which usually is the result of a query service request issued before using the modify service). This behavior will be changed in future versions of GSP.

If the `modifyType` attribute is set to use the `relation` mode of the modify service, it is possible to specify a `relationName` which identifies the relation to be modified. Only two relations may be modified by using the modify service, and the `relationType` attribute specifies which type to use in this request. When modifying a `manager` relation, it is possible to add a new manager to the relation (`addManager`) or to remove a manager from the relation (`deleteManager`). It is not possible to

remove the last manager from a `manager` relation. When modifying an `owner` relation, it is possible to add a new owner (`addOwner`) or to remove an owner from the relation (`deleteOwner`). It is not possible to remove the last owner from an `owner` relation.

```
ModifyResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  nofGsa        [1]     INTEGER,
  modifyResult  [2]     ENUMERATED {
    success         (1),
    noSuchObject    (2),
    noSuchRelation  (3),
    noPermission    (4),
    lastManager     (5),
    lastOwner       (6) } }
```

The `ModifyResponse` PDU is sent from every GSA which received the `ModifyRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `modifyResult` specifies the result of the modify request. The `success` values indicates the successful completion of the modify service, ie the requested modification has been performed and the GMS object or relation now has the new attributes.

Depending on the modify mode (either object or relation), a `modifyResult` of `noSuchObject` or `noSuchRelation` is returned if the name specified in the request does not identify an existing object or relation, or if the name specified in the request does identify an object with a different type than the one specified in the request. If a user tried to modify an object he is not allowed to modify, a `modifyResult` of `permissionDenied` will be returned. The authorization may have failed because the user's authentication level was too low (ie lower than the `authRequirements` of the object) or because the object's access policy did not give him the modify right in the `AccessRight` as defined in section 2.1.

If a user tries to remove the last manager from a `manager` relation using the `deleteManager` attribute of the modify service, an error message of `lastManager` will be given and the manager is not removed from the relation. If a user tries to remove the last owner from an `owner` relation using the `deleteOwner` attribute of the modify service, an error message of `lastOwner` will be given and the owner is not removed from the relation.

### 4.4.8 Join Group

The GAP join group service is used by a user to join a GMS group. It may also be used to join a group as a member of a group. In this case, the same authorization checks apply which would have been used when the user himself would have requested to join the group. Because it may be a time consuming process to complete a join group request (if managers have to approve or refuse the join request, depending on the `groupJoinPolicy` described in section 2.3.2), it is possible to set a timeout for manager requests. A GSA receiving a GAP join group request sends a GSP join group request to the domain where the group is stored.

```
JoinGroupRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  user                  [1]     GmsObjectName,
```

```
  userAuthLevel           [2]     AuthLevel,
  userGroups              [3]     UserGroups,
  groupName               [4]     GmsObjectName,
  joiningGroupName        [5]     GmsObjectName OPTIONAL,
  waitForManagersTimeout [6]      INTEGER OPTIONAL }
```

The `JoinGroupRequest` PDU is sent from the GSA which received the GAP join group request to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute specifies the user who wants to join a group (or who wants to join a group to a group, if the `joiningGroupName` attribute is used). The `userAuthLevel` and `userGroups` attributes contain the user's authentication level and the groups he is member of, which are necessary for authorization checking. The `groupName` attribute identifies the name of the group the user wants to join.

If the optional `joiningGroupName` attribute is used, it specifies the name of the group to be joined. Joining groups to groups is only allowed if the user is manager of both groups. The optional `waitForManagersTimeout` attribute specifies an upper bound for the time to be waited for manager responses. If this attribute is not used, no timeout has been specified by the user.

```
JoinGroupResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID           [0]     UniqueIdentifier,
  nofGsa              [1]     INTEGER,
  joinGroupResult     [2]     ENUMERATED {
    success                   (1),
    timeout                   (2),
    refuse                    (3),
    noSuchGroup               (4),
    authentificationInsuf     (5),
    notAuthorized             (6),
    alreadyMember             (7),
    alreadyIndirectMember     (8),
    joinPending               (9),
    joinPendingLimitExceeded (10) } }
```

The `JoinGroupResponse` PDU is sent from every GSA which received the `JoinGroupRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `joinGroupResult` attribute indicates the result of the join group request. If the responding GSA stores the group to be joined and successfully performed the join group operation (which implies that authorization checking was successful and possibly a number of manager requests as described in section 4.4.14), a result of `success` is returned.

A `joinGroupResult` attribute value of `timeout` will only occur if the join group request's `waitForManagersTimeout` attribute has been set and the time passed without enough manager replies to successfully join the group. A `joinGroupResult` attribute value of `refuse` indicates that too many managers refused the join request make a successful join request possible (eg, if the group's `groupJoinPolicy` is set to a `relativeQuorum` value of hundred, one refuse response from a manager is enough to make a successful join group request impossible). This `joinGroupResult` attribute value only is possible if the `waitForManagersTimeout` attribute has been set.

The noSuchGroup result indicates that the group specified in the groupName attribute of the JoinGroupRequest PDU does not exist. A authenticationInsufficient result indicates that the specified group's authRequirements are higher than the user's current AuthLevel (as given in the JoinGroupRequest PDU). A notAuthorized result indicates that the user is not authorized to join the group specified in the joiningGroupName attribute because he is not a manager of the group which shall be joined and/or not a manager of the group which shall join.

Two results are concerned with the consistency of the structure of users and groups. Both results indicate that the join group request is not processed because the user or group to be joined to a group is already member of that group. The alreadyMember result indicates that the user or group to be joined is already a direct member of the group, ie a join group request has been successfully processed before. A alreadyIndirectMember result indicates that the user or group to be joined is already an indirect member of the group, through one or more indirections.

A joinGroupResult of joinPending means that all checks which are not depending on managers have been completed successfully and that the managers will now be queried. If the GSA is not able to handle more asynchronous join group requests, it will respond with a joinPendingLimitExceeded result. It is then up to the user to decide whether to process the join group request synchronously or to wait some time and to try again with an asynchronous join group request.

### 4.4.9 Join Session

The GAP join session service is used by a user to join a GMS session. The concept of a session and its flows is described in sections 2.3.4 and 2.3.5. Because a user does not have to join all flows of a session, the flows to be joined have to be specified when the join session service is used. Because it may be a time consuming process to complete a join session request (if managers have to approve or refuse the join request, depending on the sessionJoinPolicy described in section 2.3.5), it is possible to set a timeout for manager requests. A GSA receiving a GAP join session request sends a GSP join session request to the domain where the session is stored.

```
JoinSessionRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID              [0]     UniqueIdentifier,
  user                   [1]     GmsObjectName,
  userGroups             [2]     UserGroups,
  userAuthLevel          [3]     AuthLevel,
  sessionName            [4]     GmsObjectName,
  flows                  [5]     SET OF SET {
    flowName                 [0]     GmsObjectName,
    joinRole                 [1]     ENUMERATED {
        sender                       (1),
        receiver                     (2),
        senderAndReceiver            (3) } },
  waitForManagerTimeout [6]     INTEGER }
```

The JoinSessionRequest PDU is sent from the GSA which received the GAP join session request to a domain using the domain's multicast address. The PDU contains a requestID attribute, which is used to uniquely identify this request. The user attribute specifies the user who wants to join a session. The userAuthLevel and userGroups attributes contain the user's authentication level and the groups he is member of, which are necessary for authorization checking. The sessionName attribute identifies the name of the session the user wants to join.

The flows attribute is used to specify the set of flows the user wants to join. Each of these flows is identified by a flowName and a joinRole. The joinRole attribute specifies whether the

user wants to join this flow as a `sender`, as a `receiver`, or as `senderAndReceiver`. The optional `waitForManagersTimeout` attribute specifies an upper bound for the time to be waited for manager responses. If this attribute is not used, no timeout has been specified by the user.

```
JoinSessionResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID              [0]     UniqueIdentifier,
  nofGsa                 [1]     INTEGER,
  joinSessionResult      [2]     ENUMERATED {
    success                      (1),
    timeout                      (2),
    refuse                       (3),
    noSuchSession                (4),
    noSuchFlow                   (5),
    authInsufficient             (6),
    notAuthorized                (7),
    alreadyParticipant           (8),
    joinPending                  (9),
    joinPendingLimitExceeded     (10),
    missingReceiverAddress       (11),
    missingDependencies          (12),
    flowLimitExceeded            (13) } }
```

The `JoinSessionResponse` PDU is sent from every GSA which received the `JoinSessionRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `joinGroupResult` attribute indicates the result of the join group request. If the responding GSA stores the group to be joined and successfully performed the join group operation (which implies that authorization checking was successful and possibly a number of manager requests as described in section 4.4.14), a result of `success` is returned.

A `joinSessionResult` attribute value of `timeout` will only occur if the join session request's `waitForManagerReplies` attribute has been set and the time passed without enough manager replies to successfully join the session. A `joinSessionResult` attribute value of `refuse` indicates that too many managers refused the join session request make a successful join session request possible (eg, if the session's `sessionJoinPolicy` is set to `managed` and a `relativeQuorum` value of hundred, one refuse response from a manager is enough to make a successful join session request impossible). This `joinSessionResult` attribute value only is possible if the `waitForManagerReplies` attribute has been set.

The `noSuchSession` result indicates that the session specified in the `sessionName` attribute of the `JoinSessionRequest` PDU does not exist. The `noSuchFlow` result indicates that at least one of the flows specified in the `flows` attribute of the `JoinSessionRequest` PDU does not exist. A `authenticationInsufficient` result indicates that the specified session's `authRequirements` are higher than the user's current `AuthLevel` (as given in the binding's `AuthType` described in section 2.3.1). A `notAuthorized` result indicates that the user is not authorized to join the session specified in the `sessionName`. This happens if the session's `sessionJoinPolicy` is set to `group`, but the user requesting the join is not member of the group to which the session is associated. A result of `alreadyParticipant` indicates that the user is already a participant of the session he requested to join.

A joinSessionResult of joinPending means that all checks which are not depending on managers have been completed successfully and that the managers will now be queried. If the GSA is not able to handle more asynchronous join session requests, it will respond with a joinPendingLimitExceeded result.

A joinSessionResult of missingReceiverAddress indicates that the GUA failed to provide the receiverAddress in the flowsBeingJoined attribute of the admission control response in case of joining a sender-oriented addressed flow as a receiver. This would make it impossible for the GMS to maintain a complete receiver list of this flow. Consequently, in this case the join session request is refused.

If the flows specified in the JoinSessionRequest PDU did have any dependencies relations defined (described in section 2.4.2), and the flows attribute specified a flow to join but not a flow it depends on, a joinSessionResult of missingDependencies is returned in the join session response. This result may also be returned if the flows actually being joined (as reported by in the AdmissionControlResponse PDU described below) differ from the flows in the JoinSessionRequest PDU and this caused a missing dependency. If one of the flows to be joined has reached its participantLimits (described in section 2.3.4), it is not possible to join the flow and a joinSessionResult of flowLimitExceeded is returned.

```
AdmissionControlRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID              [0]     UniqueIdentifier,
  flows                  [1]     SET OF SET {
    flowName                 [0]     GmsObjectName,
    flowAttributes           [1]     FlowAttributes,
    flowQosParameters        [2]     SET OF QosParameter,
    flowAddress              [3]     CHOICE {
      receiverOriented           [0]     OCTET STRING,
      senderOriented             [1]     SET OF OCTET STRING } } OPTIONAL,
  flowRelations          [3]     SET OF CHOICE {
    dependency               [0]     Dependency,
    synchronization          [1]     Synchronization } OPTIONAL }
```

If the authorization control has been successful (ie the requesting user is allowed to join the session), an AdmissionControlRequest PDU is sent from the GSA storing the session object to the GSA which requested the join session using a unicast connection. This happens regardless of synchronous or asynchronous mode of the join session service. The requestID identifies the join session request for which this admission control request is being sent.

The flows attribute contains a set of flows. Each flow is specified by its flowName, the flowAttributes, and flowQosParameters. These attributes are described in detail in section 2.3.4. The flowAddress attribute contains addressing information. If the flow uses receiverOriented addressing, the flowAddress simply contains the group address to join. If the flow uses senderOriented addressing, the flowAddress contains the set of receiver addresses which are necessary to join the flow as a sender. In case of joining the a senderOriented flow as a receiver, the flowAddress is not necessary and consequently not present.

The flowRelations attribute contains all relations which are used for these flows and necessary for joining them properly, ie dependency and synchronization relations. These relations are described in sections 2.4.2 and 2.4.10.

```
AdmissionControlResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID              [0]     UniqueIdentifier,
  flowsBeingJoined       [3]     SET OF SET {
```

```
    flow                    [0]     GmsObjectName,
    receiverAddress         [1]     OCTET STRING OPTIONAL } OPTIONAL }
```

The `AdmissionControlResponse` PDU is sent from the GSA which originally requested the session join to the GSA which sent the `AdmissionControlRequest` PDU using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent.

The `flowsBeingJoined` attribute of the `AdmissionControlResponse` PDU specifies which flows are eventually being joined by the user. The attribute contains a set of flows, each `flow` being identified by its name and (if the flow uses sender-oriented addressing and has been joined as `receiver` or `senderAndReceiver`) an optional `receiverAddress`, which is added to the flow's `addressingInfo` described in section 2.3.4. Because the `flowsBeingJoined` may differ from the `flows` requested in the `JoinSessionRequest` PDU, it is possible that there are missing dependencies. In this case, the `JoinSessionResponse` PDU will contain a `joinSessionResult` attribute with a value of `missingDependencies` and the join session service is cancelled, ie the user will not become a participant of the session.

### 4.4.10 Leave Session

The GAP leave session service is used to leave a session. Leaving a session means that all flows which are part of this session are left and no more data can be sent to or received from these flows. Leaving a session is pretty straightforward because no other structures depend on the participation in a session. The GSA receiving a GAP leave session PDU simply sends a GSP leave session PDU to the domain of the session object.

```
LeaveSessionRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  user                  [1]     GmsObjectName,
  sessionName           [2]     GmsObjectName }
```

The `LeaveSessionRequest` PDU is sent from the GSA which received the GAP leave session request to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute specifies the user who wants to leave a session. The `sessionName` attribute identifies the session which should be left.

```
LeaveSessionResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  nofGsa                [1]     INTEGER,
  leaveSessionResult    [2]     ENUMERATED {
    success                 (1),
    noSuchSession           (2),
    noSuchParticipant       (3) } }
```

The `LeaveSessionResponse` PDU is sent from every GSA which received the `LeaveSessionRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `leaveSessionResult` attribute indicates the result of the leave session request. If the responding GSA stores the session object and successfully performed the leave session operation (which involves local changes and the modification of remote relations), a result of `success` is returned. If the GSA does not store the session object, a `leaveSessionResult` of `noSuchSession` is returned in

the LeaveSessionResponse PDU. A noSuchMember result indicates that the session does exist, but the participant to be removed from the session is not participant of the session.

### 4.4.11 Leave Group

The GAP leave group service is used to leave a group. It may either be used to leave a group the issuing user is a member of, or to remove other users or groups from a group. In the second case, it is necessary that the user requesting the leave group service is a manager of the affected group. If a user wants to leave a group and is still participant of one or more associated sessions, he is not allowed to leave the group. The GSA receiving a GAP leave group PDU simply sends a GSP leave group PDU to the domain of the group object.

```
LeaveGroupRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID          [0]     UniqueIdentifier,
  user               [1]     GmsObjectName,
  groupName          [2]     GmsObjectName,
  leavingMemberName  [3]     CHOICE {
    user                 [0]     GmsObjectName,
    group                [1]     GmsObjectName } OPTIONAL }
```

The LeaveGroupRequest PDU is sent from the GSA which received the GAP leave group request to a domain using the domain's multicast address. The PDU contains a requestID attribute, which is used to uniquely identify this request. The user attribute specifies the user who wants to leave a group. The groupName attribute identifies the group which should be left. If not the user himself wants to leave the group, but he wants to remove another member from the group, the optional leavingMemberName is specified. It specifies whether a user or a group shall be removed from the group.

```
LeaveGroupResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID          [0]     UniqueIdentifier,
  nofGsa             [1]     INTEGER,
  leaveGroupResult   [2]     ENUMERATED {
    success                  (1),
    noSuchGroup              (2),
    noSuchMember             (3),
    notManager               (4),
    activeSessionWithinGroup (5) } }
```

The LeaveGroupResponse PDU is sent from every GSA which received the LeaveGroupRequest PDU to the requesting GSA using a unicast connection. The requestID attribute uniquely identifies the request for which this response is being sent. The nofGSA attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The leaveGroupResult attribute indicates the result of the leave group request. If the responding GSA stores the group object and successfully performed the leave group operation (which involves local changes and the modification of remote relations), a result of success is returned.

A noSuchGroup result indicates that the groupName specified in the LeaveGroupRequest PDU does not exist. A noSuchMember result indicates that the group does exist, but the member to be removed from the group (either the requesting user or the member specified by the leavingMember-Name attribute) is not member of the group. If the leavingMemberName attribute has been used in

the request, but the requesting user is not manager of the group, a `notManager` result is returned. Finally, the `activeSessionWithinGroup` result indicates that the group may not be left because there are active sessions associated with that group which have to left first.

### 4.4.12 Delete

The GAP delete service is used to delete an object from the GMS. Because most object types (except flow templates and certificates) are used for many other services, only objects which are in a certain state (ie there are no related objects which might be affected by deleting the object) may be deleted. A GSA receiving a GAP delete request sends a GSP delete request to the domain where the object is stored.

```
DeleteRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  user          [1]     GmsObjectName,
  userAuthLevel [2]     AuthLevel,
  userGroups    [3]     UserGroups,
  objectName    [4]     GmsObjectName }
```

The `DeleteRequest` PDU is sent from the GSA which received the GAP delete request to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `user` attribute specifies the user who wants to delete an object. The `userAuthLevel` and `userGroups` attributes contain the user's authentication level and the groups he is member of, which are necessary for authorization checking. The `objectName` attribute identifies the name of the object the user wants to delete.

```
DeleteResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID     [0]     UniqueIdentifier,
  nofGsa        [1]     INTEGER,
  deleteResult  [2]     ENUMERATED {
    success             (1),
    noSuchObject        (2),
    permissionDenied    (3),
    existingRelations   (4) } }
```

The `DeleteResponse` PDU is sent from every GSA which received the `DeleteRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `deleteResult` attribute indicates the result of the delete request. If the responding GSA stores the object to be deleted and successfully performed the delete operation (which implies that authorization checking was successful), a result of `success` is returned.

If the `deleteResult` is `noSuchObject`, the `objectName` specified in the `DeleteRequest` PDU does not refer to an existing object. If the result is `permissionDenied`, the object exists, but the user is not authorized to delete the object. The authorization may have failed because the user's authentication level was too low (ie lower than the `authRequirements` of the object) or because the object's access policy did not give him the delete right in the `AccessRight` as defined in section 2.1. If the object which should be deleted is still in use, the `deleteResult` will be `existingRelations` meaning that there are still relations between the object to be deleted and other objects which should be deleted using other GMS services before deleting the object will be successful.

### 4.4.13 Renegotiate

The GAP renegotiate service is used to change one or more QoS parameters of a flows. The concept of a QoS renegotiation is described in section 2.2. GMS currently does not provide support to actually carry out a renegotiation in terms of exchanging proposed values, collecting answers and then deciding which new QoS value to choose. In the current version of GSP, it is only possible to propagate a set of new QoS values to all flow participants. The GSA receiving a GAP renegotiate PDU simply sends a GSP group session PDU to the domain of the flow object.

```
RenegotiateRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  requestingUser        [1]     GmsObjectName,
  flow                  [2]     GmsObjectName,
  renegotiatedQosParam  [3]     SET OF QosRenegotiation }
```

The `RenegotiateRequest` PDU is sent from the GSA which received the GAP renegotiate request to a domain using the domain's multicast address. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `requestingUser` attribute specifies the user who wants to renegotiate a flow. The `flow` attribute identifies the flow which is subject of the QoS renegotiation. The `renegotiatedQosParam` attribute is a set of `QosRenegotiation` values, which are described in section 2.2. This attribute defines the set of QoS parameters which should be set to new values.

```
RenegotiateResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  nofGsa                [1]     INTEGER,
  renegotiateResult     [2]     ENUMERATED {
    success                     (1),
    noSuchFlow                  (2),
    notAuthorized               (3),
    notRenegotiable             (4),
    noSuchQosParameter          (5),
    nameTypeMissmatch           (6),
    illegalRenegotiationValues  (7) } }
```

The `RenegotiateResponse` PDU is sent from every GSA which received the `RenegotiateRequest` PDU to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `nofGSA` attribute contains the number of GSAs in the responding GSA's domain (taken from the responding GSA's GSA table) and is used by the receiving GSA to determine whether all GSAs of the requested domain have responded.

The `renegotiateResult` attribute indicates the result of the renegotiation request. If the responding GSA stores the flow object and successfully performed the renegotiation operation (which only involves local changes), a result of `success` is returned. Depending on the flow's renegotiation policy (described in section 2.3.4), renegotiation PDUs (described in section 4.4.17) will be sent to senders, receivers, or managers of the flow.

Possible error messages of the `renegotiateResult` are as follows. `noSuchFlow` indicates that the `flow` specified in the `RenegotiateRequest` PDU does not exist. The `notAuthorized` result indicates that the requesting user is not authorized to perform a QoS renegotiation for that flow (the authorization check is based on the users identity and the `renegotiation` attribute of the flow's `FlowAttributes` described in section 2.3.4). If the flow is not renegotiable at all, the `notRenegotiable` result is returned.

The last three results refer to the `renegotiatedQosParameters` in the `RenegotiateRequest`. The `noSuchQosParameter` result indicates that a `QosRenegotiation` with a non-existing QoS parameter has been specified. The `nameTypeMismatch` indicates that a `QosRenegotiation` contained an existing parameter name, but specified the wrong type for that parameter. Finally, the `illegalRenegotiationValues` result is returned if the new values specified in a `QosRenegotiation` lie outside the renegotiation limits for this parameter.

### 4.4.14 Manager

The GAP manager service is used by the GMS to query managers (of groups or sessions) about their response to join requests issued by users. If a GSA getting a GAP join group or join session request determines that it is necessary to query one or more managers about the join request, it uses the GSP manager service to send these requests to the GSAs to which the managers are bound.

```
ManagerRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  manager               [1]     GmsObjectName,
  managerRequestType    [2]     CHOICE {
    joinGroupRequest        [0]     SET {
      joinedGroup               [0]     GmsObjectName,
      memberCandidate           [1]     CHOICE {
        user                        [0]     GmsObjectName,
        group                       [1]     GmsObjectName } },
    joinSessionRequest      [1]     SET {
      joinedSession             [0]     GmsObjectName,
      participantCandidate      [1]     GmsObjectName } } }
```

The `ManagerRequest` PDU is sent from the GSA to which the user requesting the join group or join session is bound to the GSA to which the manager is bound using a unicast connection. The PDU contains a `requestID` attribute, which is used to uniquely identify this request. The `manager` attribute specifies the manager which should be requested to approve or refuse the join request.

The `managerRequestType` determines whether the manager is requested because of a join group or because of a join session request. If it is set to `joinGroupRequest`, the manager request contains a `joinedGroup` specifying the group to be joined and a `memberCandidate` attribute, which specifies the `user` or the `group` requesting the join.

If the `managerRequestType` attribute is set to `joinSessionRequest`, the manager request contains a `joinedSession` specifying the session to be joined and a `participantCandidate` attribute, which specifies the user who wants to join the session.

```
ManagerResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  requestID             [0]     UniqueIdentifier,
  managerResult         [1]     ENUMERATED {
    approve                 (1),
    noMorePendingRequests   (2),
    refuse                  (3) } }
```

The `ManagerResponse` PDU is sent from the GSA to which the manager is bound to the requesting GSA using a unicast connection. The `requestID` attribute uniquely identifies the request for which this response is being sent. The `managerResult` specifies the result of the manager query, ie contains the response the manager gave using the GAP manager service. It may either specify

the manager's approval (`approve`) or refusal (`refuse`) of the join request. A third result value is `noMorePendingRequests`, which is signaling that the manager's GUA can currently not handle more outstanding manager requests.

### 4.4.15 Notification

The GAP notification service is used to inform users about certain events. Depending on the content of the notification service PDU, a number of events can be notified. Basically, there are two classes of notifications (which may contain more than one notification type). The first class informs users about group members binding or unbinding, changes in the members set, or new or deleted associated sessions of a group they are a member of. The second class informs users about changes in the participants set of a session they are a participant of. The GSP notification service is used to inform a GSA that a user bound to it should be notified using the GAP notification service.

```
NotificationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  notifiedUser          [1]     GmsObjectName,
  notificationType      [2]     CHOICE {
    bindNotification        [0]     SET {
      userName                  [0]     GmsObjectName,
      groupName                 [1]     GmsObjectName },
    unbindNotification      [1]     SET {
      userName                  [0]     GmsObjectName,
      groupName                 [1]     GmsObjectName },
    joinGroupNotification   [2]     SET {
      joinedGroup               [0]     GmsObjectName,
      newMember                 [1]     GmsObjectName },
    leaveGroupNotification  [3]     SET {
      leftGroup                 [0]     GmsObjectName,
      leavingMember             [1]     GmsObjectName },
    createSessionNotification [4]   SET {
      groupName                 [0]     GmsObjectName,
      associatedSession         [1]     GmsObjectName },
    deleteSessionNotification [5]   SET {
      groupName                 [0]     GmsObjectName,
      associatedSession         [1]     GmsObjectName },
    joinSessionNotification [6]     SET {
      joinedSession             [0]     GmsObjectName,
      newParticipant            [1]     GmsObjectName },
    leaveSessionNotification [7]    SET {
      leftSession               [0]     GmsObjectName,
      leavingParticipant        [1]     GmsObjectName } } }
```

The `NotificationRequest` PDU is sent to the GSA to which the user who shall receive the `NotificationRequest` is bound using a unicast connection. The `notifiedUser` attribute specifies the name of the user to be notified. The `notificationType` specifies which type of event occured and will be described according to the two available classes of notifications.

The first class of notifications is associated with the group membership of a user. Each group has a `groupNotificaPolicy` attribute (described in section 2.3.2) which describes the events which should be notified, and whether `managers`, `members`, or `managersAndMembers` should be notified. A `bindNotification` notifies a user that a member (specified by the `userName`) of the group `groupName`

successfully bound to the GMS. A `unbindNotification` notifies a user that a member (specified by the `userName`) of the group `groupName` unbound from the GMS. A `joinGroupNotification` notifies about a successful join to the group `joinedGroup` of the new member `newMember`. A `leaveGroupNotification` notifies a user that the group member `leavingMember` left the group `leftGroup`. A `createSessionNotification` notifies a user that a new session `associatedSession` has been associated with the group `groupName`. A `deleteSessionNotification` notifies a user that the session `associatedSession` which has been associated with `groupName` has been deleted. All these notifications are only sent to a user if he is member or manager of the affected group and the group's `groupNotificaPolicy` attribute is set accordingly.

The second class of notifications informs users about changes in the participants set of a session they are a participant of. Each session has a `sessionNotifiPolicy` attribute (described in section 2.3.5) which describes the events which should be notified, and whether `managers`, `participants`, or `managersAndParticip` should be notified. A `joinSessionNotification` notifies about a successful join to the session `joinedSession` of the new participant `newParticipant`. A `leaveSessionNotification` notifies a user that the session participant `leavingParticipant` left the session `leftSession`. All these notifications are only sent to a user if he is participant or manager of the affected session and the session's `sessionNotifiPolicy` attribute is set accordingly.

### 4.4.16 Invitation

The GAP invite service is used to invite another user to a specific event, which will usually be an application using a session. The GSA to which the inviting user is connected will forward the invitation to the invited user using the GSP invitation service.

```
InvitationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  invitedUser           [1]     GmsObjectName,
  invitingUser          [2]     GmsObjectName,
  invitationType        [3]     CHOICE {
    gmsInviteMessage        [0]     IA5String,
    appInviteMessage        [1]     SET {
        application             [0]     Application,
        message                 [1]     OCTET STRING} } }
```

The `InvitationRequest` PDU is sent to the GSA to which the user who shall receive the `InvitationRequest` is bound using a unicast connection. The `invitedUser` attribute specifies the user who is invited. The attribute `invitingUser` gives the name of the user who issued the invitation (using the GAP invite service). The `invitationType` of an invitation request may either be a `gmsInviteMessage`, which is a simple `IA5String`, or an `appInviteMessage`. In this case, the `invitationType` contains the `application` and the `message` as an application specific octet string.

### 4.4.17 Renegotiation

The GAP renegotiation service is used by the GSA to inform users about a QoS renegotiation. The trigger for a renegotiation service request always is a renegotiate request (described in section 4.4.13) which modified a flow's QoS parameters. Renegotiation requests are only sent if the flow's `renegotiation` attribute (described in section 2.3.4) is set accordingly. Renegotiation requests may be sent to senders and/or receivers of a flow.

```
RenegotiationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
  notifiedUser          [0]     GmsObjectName,
  sessionName           [1]     GmsObjectName,
```

```
  flowName              [2]     GmsObjectName,
  renegotiatedQosParam  [3]     SET OF QosRenegotiation }

END
```

The `RenegotiationRequest` PDU is sent to the GSA to which the user who shall receive the `RenegotiationRequest` is bound using a unicast connection. The `notifiedUser` attribute specifies the user who is notified of the flow's renegotiation. The `sessionName` and `flowName` attributes specify the flow which is renegotiated and the session it is in. The `renegotiatedQosParam` attribute is a set of `QosRenegotiation` attributes (described in section 2.2) which specify the new QoS values. It is the same set which was specified in the `renegotiatedQosParam` attribute of the renegotiate service request (`RenegotiateRequest`) described in section 4.4.13.

# References


[1] 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.

[2] 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.

[3] R. Braden. Extending TCP for Transactions – Concepts. Internet RFC 1379, November 1992.

[4] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. Internet RFC 1644, July 1994.

[5] Andrew Campbell, Geoff Coulson, and David Hutchison. A Multimedia Enhanced Transport Service in a Quality of Service Architecture. In D. Shepherd, G. Blair, G. Coulson, N. Davies, and F. Garcia, editors, *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, volume 846 of *Lecture Notes in Computer Science*, pages 124–137, Lancaster, UK, November 1993. Springer-Verlag.

[6] Pascal Freiburghaus. System-Protokollarchitektur für das Group and Session Management System. Master's thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, September 1996.

[7] Walter Gora and Reinhard Speyerer. *Abstract Syntax Notation One (ASN.1).* DATACOM, Bergheim, Germany, second edition, 1990.

[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] International Organization for Standardization. Information technology – Generic coding of moving pictures and associated audio information. ISO/DIS 13818, 1995.

[10] International Telecommunication Union. Specification of Abstract Syntax Notation One (ASN.1). Recommendation X.208, 1988.

[11] International Telecommunication Union. Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). Recommendation X.209, 1988.

[12] International Telecommunication Union. The Directory – Overview of Concepts, Models and Services. Recommendation X.500, March 1995.

[13] Daniel Koller. System-Protokolloperationen für das Group and Session Management System. Master's thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, September 1996.

[14] P. Mockapetris. Domain Names – Concepts and Facilities. Internet RFC 1034, November 1987.

[15] Gerhard Nigg. Multicast-Module für Da CaPo. Master's thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, March 1996.

[16] Michael Sample. Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler. Technical report, University of British Columbia, Vancouver, July 1993.

[17] 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.

[18] Douglas Steedman. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference.* Technology Appraisals, Twickenham, UK, 1993.

[19] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications, and Applications.* Prentice-Hall, Upper Saddle River, New Jersey, 1995.

[20] W. Richard Stevens. *Unix Network Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[21] 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.

[22] Erik Wilde, Pascal Freiburghaus, Daniel Koller, and Bernhard Plattner. A Group and Session Management System for Distributed Multimedia Applications. In *Multimedia Telecommunications and Applications – Proceedings of the Third COST 237 Workshop*, Barcelona, Spain, November 1996.

[23] 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.