Specification of GMS Access Protocol (GAP) 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 a specification of GAP, the GMS access protocol. GAP is specified by state diagrams describing the behavior of two communication entities, 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 GAP, the GMS access protocol, version 1.0, which is used to access the group and session management system (GMS). The general architecture of GMS is shown in figure 2. 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 major benefits of the GMS architecture are 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 [1]). 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.

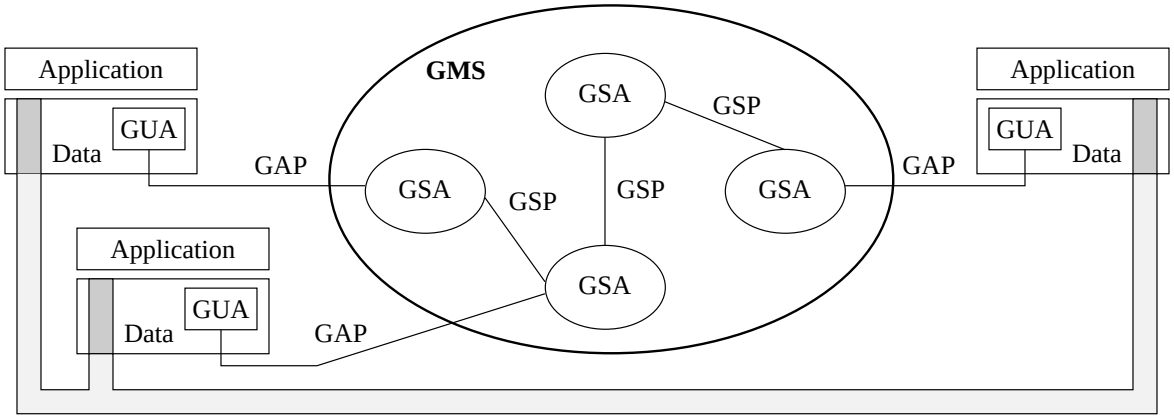

Figure 1: GMS architecture

To understand GAP, 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 GAP (described in section 3.2) are specified in ASN.1 as defined by the ITU [2, 5, 11]. Because ASN.1 only describes the syntax, the specifications are accompanied by explanations of the semantics. The "`--snacc isPdu:"TRUE" --`" comments are necessary for the snacc ASN.1 compiler which is described by Sample and Neufeld [9, 10]. 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.

A closer look at GMS and its architecture (which is, in a very general way, depicted in figure 1) shows that the main component for a GMS user is a GUA. The GUA implements GAP and provides an API which may be used to access GMS. Figure 2 shows how a GUA may be integrated into a communications platform. The functionality of the communications platform is entirely outside the scope of GMS, which just provides support for group and session management. Furthermore, the design of the GUA's API also is not in the scope of this specification. It is only described which data is exchanged between a GUA and a GSA and according to which rules this exchange is performed. The actual implementation of a GUA also is outside the scope of this specification. Any software component implementing GAP may be viewed as a GUA, even if it does not conform to the modular design shown in figure 2.

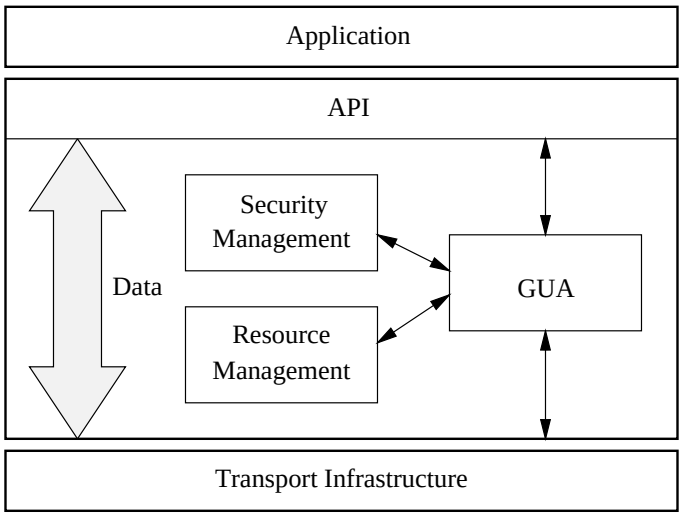

Figure 2: Group and session management inside a communications platform

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 GAP 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 GAP PDUs (described in section 3.2).

```
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). In order to use this information with GAP operations, the `Application` data type is also used in PDUs for binding and unbinding applications (described in sections 3.2.3 and 3.2.4), invite operations (described in section 3.2.16), and invitations (described in section 3.2.19) being received. 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. A more detailed description of the authentication concept and procedure is given in section 3.2.2.

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 [8] or the concept of distinguished names of the ITU's X.500 directory [7]). 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, the `FlowTemplate` object type (described in section 2.3.3) to specify for which transport service a flow template may be used, and the bind user service (described in section 3.2.2), where the binding user has to specify which `TransportService` he is using. 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. 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 renegotiate service (described in section 3.2.15) and the renegotiation service (described in section 3.2.20), 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 (in sections 3.2.3 and 3.2.4 it is described how application binding is handled).

```
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 (described in detail in sections3.2.10, 3.2.11 and 3.2.17).

`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.

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` decides 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, the `relativeQuorum` 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. 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 renegotiation request described in section 3.2.20). 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 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. This set is automatically updated when users 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. A description on how this information may be used can be found in session 3.2.3.

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 delete service described in section 3.2.14.

```
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. This procedure is described in more detail in sections 2.3.5 and 3.2.11.

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 [3], 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 modify service described in section 3.2.9), 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 modify service described in section 3.2.9), 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 (during the create service described in section 3.2.7), 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 delete service described in section 3.2.14).

```
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 (described in detail in section 3.2.11). The `Receiver` relation is especially important for dealing with renegotiations, because all receivers of a flow must get a renegotiation notification (described in section 3.2.20) when a QoS renegotiation for this flow is initiated using the renegotiate service described in section 3.2.15.

```
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 (described in detail in section 3.2.11). The `Sender` relation is especially important for dealing with renegotiations, because all senders of a flow must get a renegotiation notification (described in section 3.2.20) when a QoS renegotiation for this flow is initiated using the renegotiate service described in section 3.2.15.

```
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 [12].

```
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 GMS Access Protocol (GAP)

GAP is specified in two parts, one defining the behavior of the communicating entities, the other defining the data being exchanged. The behavior is defined using finite state machines, which are described in section 3.1. The exchanged data is defined in ASN.1 PDUs, which are listed and explained in section 3.2. Both sections together form a complete description of GAP. The semantics of each operation is described in the corresponding subsection of section 3.2.

3.1 State diagrams

The finite state machines defining the behavior of both the GUA and GSA are defined graphically. State transitions are depicted by arrows which are labeled with events and actions, separated by a slash. State transitions which may occur without an event causing them have empty events (no text before the slash). Events and actions are denoted with a syntax derived from the notation of ITU recommendation X.227 [4]. In this notation, the suffix determines an event's or action's meaning, with Req standing for requests, Ind standing for indications, Res standing for responses, and Cnf standing for confirmations. PDUs are denoted accordingly, RQ are request, and RE are response PDUs. Responses (Res or RE) and confirmations (Cnf) may also be followed by a + or - sign, indicating a positive or negative result. If no such sign is given, then positive and negative responses or confirmations are referred to.

The state diagrams are defined hierarchically, ie the GSA and GUA finite state machines are each defined by three hierarchically organized diagrams. States which contain a number of other states (also referred to as compound states) are denoted by double bounded boxes. The initial state of each diagram (ie the state in which the execution of this diagram starts) is marked by a bold outline. Furthermore, there are parallel state machines, which are denoted by a dashed line which separates the state machines which should be executed in parallel. All parallel state machines start execution in their initial state if a transition leads into the parallel execution box. All parallel state machines are exited if one of the parallel state machines executes a transition which points to a state outside the parallel execution box.

3.1.1 GSA

The GSA side of the protocol is specified in three state diagrams (figures 3 to 5). The separation into three diagrams reflects the three major phases of GAP. The first diagram (figure 3) shows the connection phase between GUA and GSA. The second state diagram (figure 4) defines the authentication phase for a user trying to bind to the GMS, while the third diagram (figure 5) shows, how the GMS can be used for information exchange once a user has successfully bound to it.

First, the GSA is in the idle state. A GUA may now send a BindGuaRQ PDU (described in section 3.2.1) which will either be answered with a negative response (BindGuaRE-), causing the GSA to stay in in the idle state, or with a positive response (BindGuaRE+), resulting in a transition to the parallel execution box.

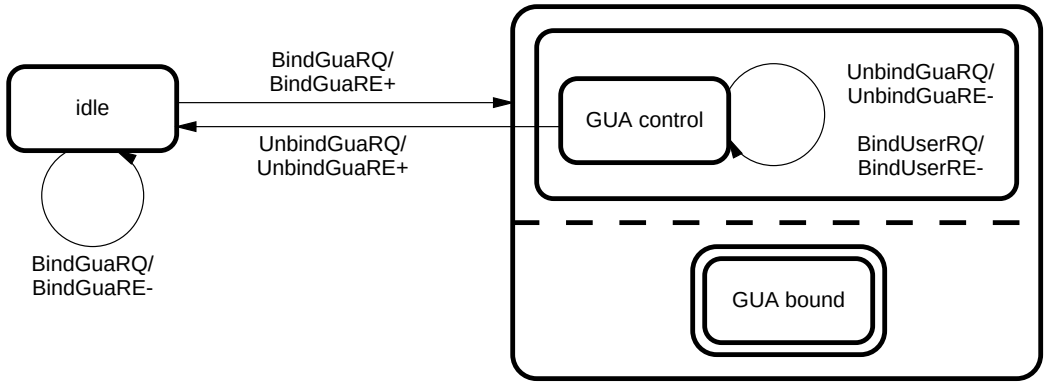

Figure 3: GAP/GSA state diagram

The parallel execution box contains one state machine consisting only of the GUA control state (which controls the GSA-GUA connection) and one or more parallel instances of the GUA bound compound state (which is specified in figure 4). Each instance of a GUA bound state machine can handle a binding for one user, so the number of parallel GUA bound instances is equivalent to the number of user bindings which may be active for one GUA-GSA connection in parallel. Each GUA bound state machine is identified by a `BindID` (which is explained in section 3.2). If an incoming BindUserRQ can not be assigned to a free GUA bound state machine (ie a GUA bound state machine which is in the GUA bound state), it is rejected (BindUserRE-) with a result code indicating that no more users are allowed to bind (see section 3.2.2 for details). This rejection is performed by the GUA control state machine.

An UnbindGuaRQ indicates that the GUA wants to terminate the GAP connection. Based on the state of the GUA bound state machines (indicating whether there are any users bound at the moment) and on the settings of the `forceUnbind` attribute of the UnbindGuaRQ PDU (described in section 3.2.6), the request is either rejected (UnbindGuaRE-), leaving the GSA in the GUA control

state and the parallel execution box (thus preserving all GUA bound state machines), or accepted (UnbindGuaRE+), exiting the parallel execution box (and consequently all GUA bound state machines) and making the transition to the idle state.

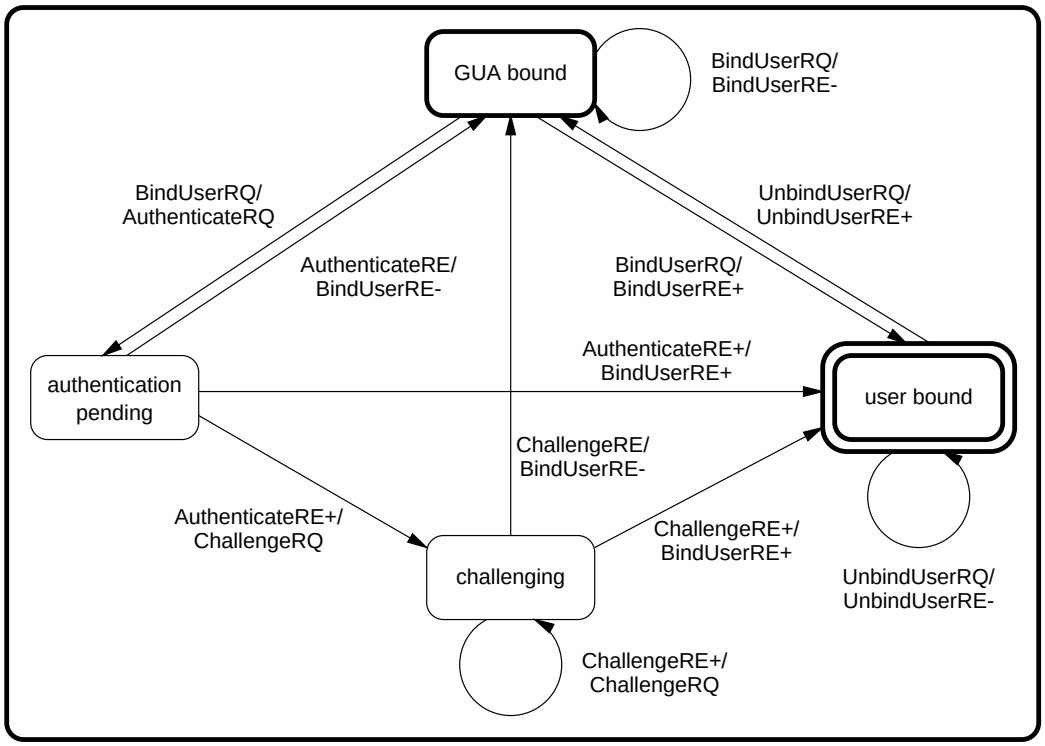

Figure 4: GAP/GSA state diagram (GUA bound compound state)

Figure 4 depicts the GUA bound state machine, which is used for the authentication of users. For every user trying to bind to the GMS, one instance of this state machine exists (if there are no more instances and a BindUserRQ is received, it is rejected by the GUA control state machine, as described before). Each instance is identified through the `TmpBindID` during authentication. If authentication was successful, the state machine makes a transition to the user bound compound state (which is specified in figure 5) and the user is assigned a `BindID`. Depending on the type of authentication defined for a particular user (described in section 2.3.1), a BindUserRQ PDU (if it is not answered with a BindUserRE- PDU, which does not change the GUA bound state machine's state) is answered with a BindUserRE+ PDU and may either make the transition to the authentication pending state, meaning that further authentication is necessary and requesting the respective information by sending an AuthenticateRQ PDU, or go directly to the user bound compound state, meaning that no authentication is necessary (which is true for anonymous users bindings and users which accept authentication level none as described in section 2) and the bind user request is confirmed with a BindUserRE+ PDU.

If authentication is required (ie the GSA in in the authentication pending state), a list of available and supported authentication schemes is sent to the GUA in the AuthenticateRQ PDU. The GUA may then respond with a AuthenticateRE- PDU, meaning that the authentication procedure is aborted. In this case, the GSA sends back a BindUserRE- PDU with the appropriate result (described in section 3.2.2) and again enters the GUA bound state. An AuthenticateRE+ PDU sent by the GUA may yield one of three transitions. If the authentication information contained in the PDU is incorrect, the authentication fails and a BindUserRE- PDU is generated by the GSA (which then makes the transition back to the GUA bound state). If the authentication information is correct and sufficient

to successfully authenticate the user (such as a correct password), then a BindUserRE+ PDU is sent to the GUA and the GUA bound state machine enters the user bound compound state. If the authentication information is correct but not sufficient for successful authentication, a ChallengeRQ PDU is sent to the GUA which contains challenge data which is necessary for a successful authentication. In this case, the GSA's GUA bound state machine enters the challenging state. The GUA's response may either be negative (ChallengeRE-, yielding a BindUserRE- PDU being sent back and making the transition to GUA bound) or positive (ChallengeRE+), in which case the authentication data may have been incorrect (also yielding a BindUserRE- PDU being sent back and making the transition to GUA bound), correct but needing more challenges to be exchanged (staying in the challenging state and generating the next ChallengeRQ), or correct and sufficient (responding with a BindUserRE+ PDU and making the transition to the user bound compound state). A more detailed description of the authentication procedures is given in section 3.2.2. The last step in the authentication process is always a BindUserRE PDU, which either makes the transition to the GUA bound state (unsuccessful authentication) or to the user bound compound state (successful authentication).

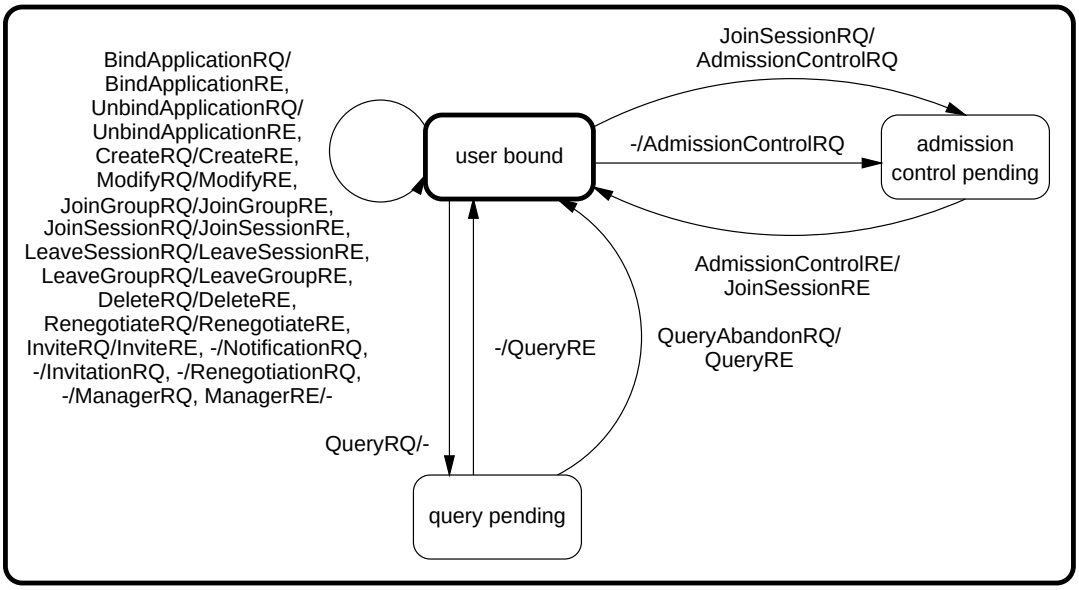

Figure 5: GAP/GSA state diagram (user bound compound state)

Figure 5 shows the specification of the user bound compound state. For every user who successfully bound to the GMS, there is one instance of this state machine, which is identified through the `BindID` in exchanged PDUs. Most requests from the GUA are handled directly, ie the request PDU is received, processed, and the result is sent back in a response PDU. There are also four PDUs which may be sent without any event being generated by the GUA (ManagerRQ, NotificationRQ, InvitationRQ, and RenegotiationRQ), because in these cases the GSA becomes active because of events being generated by other GUAs or GSAs. A ManagerRE PDU may be received at any time as an answer to a former ManagerRQ (a `ManagerRequestID` is used to associate requests and responses).

Because queries (described in section 3.2.8) may take a long time to complete, it is possible to abandon a query started by a QueryRQ PDU from the GUA by sending a QueryAbandonRQ PDU. After receiving the QueryRQ PDU, the GSA goes into the query pending state. This state can either be left by abandoning the query (as described before) or it is left when the query is completed and the GSA sends the query results in a QueryRE PDU. In both cases, the GSA goes back into the user bound state.

Joining a session (as described in section 3.2.11) may be done in two different ways. Depending

on how `waitForManagerReplies` is set in the JoinSessionRQ PDU (described in section 3.2.11), the GSA behaves differently. If `asynchronousTimeout` is selected, then the GSA responds with a JoinSessionRE PDU containing a `pendingJoinId`, which is used to refer to the JoinSessionRQ. If the managers' replies confirmed the JoinSessionRQ (depending on the session's `sessionJoinPolicy` as described in section 2.3.5), then the GSA send a AdmissionControlRQ PDU and enters the admission control pending state. If the managers' replies did not allow the new participant to join the session, the GSA uses a NotificationRQ with the `pendingJoinNotification` attribute set to either notify a `timeout` or a `failure`. The admission control pending state is also entered if the GUA set the `synchronousTimeout` attribute in the JoinSessionRQ PDU. In this case, the GSA waits for the managers' replies and then either uses a JoinSessionRE PDU to reject the join (and remain in the user bound state) or an AdmissionControlRQ PDU and enters the admission control pending state. The GUA uses the information of the AdmissionControlRQ PDU to perform a local admission control (checking whether the local resources are sufficient to join the session) and then uses an AdmissionControlRE PDU to inform the GSA of the result of the admission control. Making the transition back to the user bound state, the GSA then uses a JoinSessionRE PDU to either accept the join or to inform the GUA of possible problems. Only after a JoinSessionRE+, the GUA may actually perform the session join, ie join all the flows which it selected from the session's set of flows.

3.1.2 GUA

According to the GSA side of GAP, the GUA side's behavior is also specified in three state diagrams (figures 6 to 8). These state diagrams are also separated into different phases of GAP communications, the first (figure 6) showing the connection establishment between GUA and GSA, the second (figure 7) showing the authentication of a user, and the third (figure 8) defining the procedures once a user is successfully bound. The GUA state machines differ from the GSA state machines in having API events (Req and Res) and actions (Ind and Cnf) which stand for down-calls resp. up-calls at the interface.

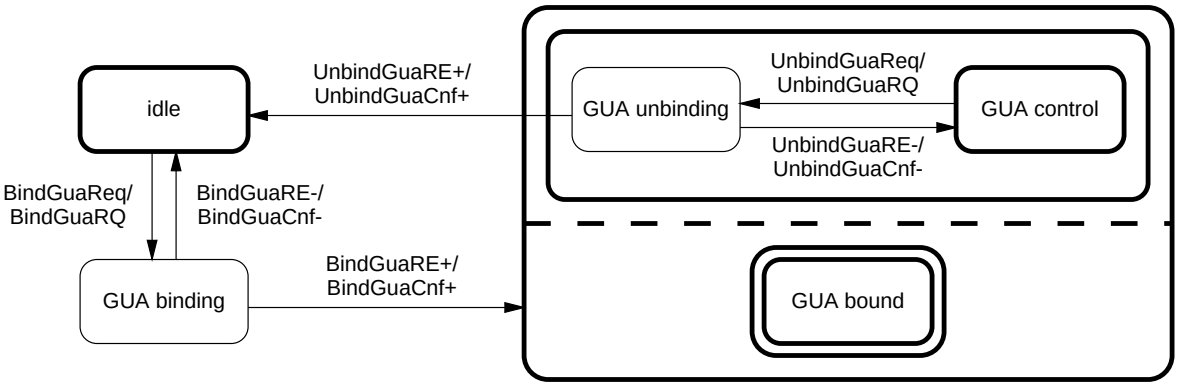

Figure 6: GAP/GUA state diagram

The GUA starts in the idle state when it is initiated. The GUA user can then issue a BindGuaReq, which will cause the GUA to send a BindGuaRQ PDU (described in section 3.2.1) to the GSA. With this PDU, it can request a binding to the GMS. Waiting for the GSA's response, the GUA is in the GUA binding state. If the GSA rejects the BindGuaRQ with a BindGuaRE-, a BindGuaCnf- is generated at the API and the GUA goes back to idle. If the BindGuaRQ is positively confirmed by the GSA by responding with a BindGuaRE+, a BindGuaCnf+ is generated and the GUA will make the transition to the parallel execution box.

Inside the parallel execution box, there are at least two state machines running in parallel. The

first one is the GUA control state machine consisting of the GUA control state and the GUA unbinding state. Additionally, there is one or more parallel instances of the GUA bound compound state (which is specified in figure 5). Each instance of a GUA bound state machine can handle a binding for one user, so the number of parallel GUA bound instances is equivalent to the number of user bindings which may be active for one GUA-GSA connection in parallel. Each GUA bound state machine is identified by a `BindID` (which is explained in section 3.2).

The GUA control state machine initially is in the GUA control state, which is only left when a UnbindGuaReq is issued, which will cause the GUA to send a UnbindGuaRQ to the GSA. Depending on the state of the GUA (active user bindings) and the `forceUnbind` attribute of the UnbindGuaRQ PDU (described in section 3.2.6), the request is rejected by the GSA (UnbindGuaRE-), causing the generation of a UnbindGuaCnf- and a transition back to the GUA control state, or confirmed by the GSA (UnbindGuaRE+), thus exiting the parallel execution box (and consequently all GUA bound state machines) and making the transition to the idle state while generating a UnbindGuaCnf+.

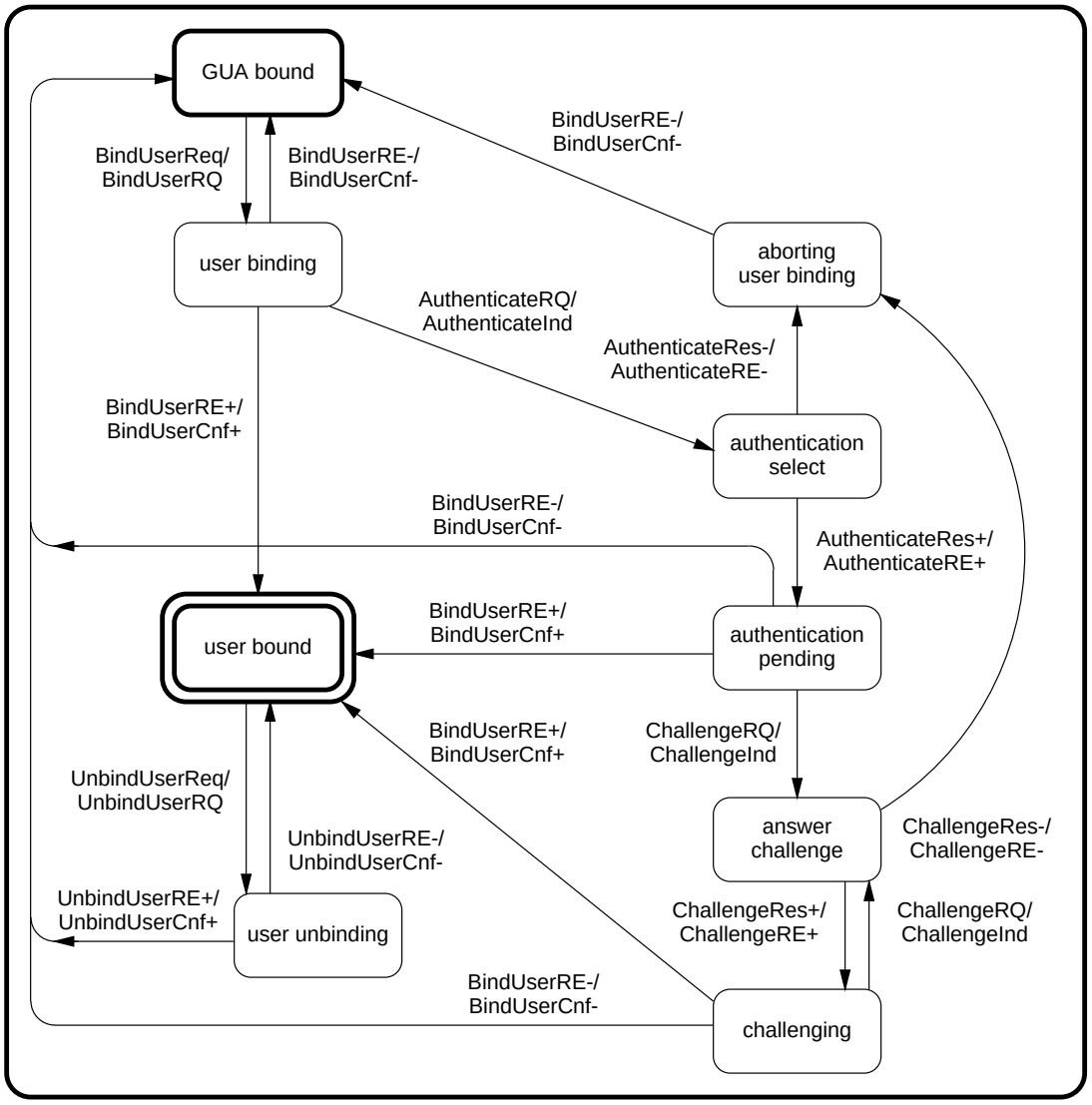

Figure 7: GAP/GUA state diagram (GUA bound compound state)

Figure 7 depicts the GUA bound state machine, which is used for the authentication of users. For every user trying to bind to the GMS, one instance of this state machine must exist. The

identification of this state machine is done with the TmpBindID described in section 3.2. If a user issues a BindUserReq and there is no free instance (ie an instance which is in the GUA bound state), the BindUserReq has to be rejected locally, ie at the GUA's API. If there is a free instance, the GUA sends a BindUserRQ PDU to the GSA and enters the user binding state. If the BindUserRQ is rejected (ie a BindUserRE- PDU is received), the GUA generates a BindUserCnf- and goes back to the GUA bound state. If the GSA responds with a BindUserRE+ PDU, the authentication phase is finished, the GUA generates a BindUserCnf+ and enters the user bound compound state. Otherwise, an AuthenticateRQ PDU is received, which contains one or more possible ways to authenticate the user. Generating an AuthenticateInd, the GUA then enters the authentication select state. If the user decides to abort the authentication by issuing a AuthenticateRes-, the GUA sends a AuthenticateRE- PDU to the GSA, enters the aborting user binding state, and waits for a BindUserRE- PDU to be sent back by the GSA. Then the GUA goes to the GUA bound state and generates a BindUserCnf-.

Being in the authentication select state, the user must now select an authentication method and use an AuthenticateRes+ to pass this selection together with authentication information to the GUA. The GUA then sends an AuthenticateRE+ PDU to the GSA and enters the authentication pending state. If the authentication information was invalid, the GSA responses with a BindUserRE- PDU, which causes the GUA to generate a BindUserCnf- and to go back to the GUA bound state. If the authentication information sent in the AuthenticateRE+ PDU was correct and sufficient for successful authentication, the GSA responses with a BindUserRE+ PDU, which causes the GUA to generate a BindUserCnf+ and to enter the user bound compound state. If the authentication information sent in the AuthenticateRE+ PDU was correct but not sufficient, the GSA responses with a ChallengeRQ PDU which contains challenge data which is necessary for a successful authentication. This PDU generates a ChallengeInd and a transition to the answer challenge state. If the user decides to cancel the authentication at this point, a ChallengeRes- is issued which causes the GUA to send a ChallengeRE- and move to the aborting user binding state. The GSA responds with a BindUserRE- PDU, which causes the GUA to generate a BindUserRes- and go back to the GUA bound state.

If the user decides to answer the challenge, a ChallengeRE+ is issued, which causes the GUA to make the transition to the challenging state and send a ChallengeRE+ PDU to the GSA. If the answer was incorrect, the GSA responds with a BindUserRE- PDU, which generates a BindUserCnf- and takes the GUA back to the GUA bound state. If the answer was correct and sufficient for successful authentication, the GSA responses with a BindUserRE+ PDU, which causes the GUA to generate a BindUserCnf+ and to enter the user bound compound state. If the challenge answer sent in the ChallengeRE+ PDU was correct but not sufficient, the GSA responses with a ChallengeRQ PDU which contains the next challenge. This PDU generates another ChallengeInd and a transition back to the answer challenge state. This scheme continues, until either a BindUserRE+ or a BindUserRE- PDU is received. A more detailed description of the authentication procedures is given in section 3.2.2.

Figure 8 shows the specification of the user bound compound state. For every user who successfully bound to the GMS, there is one instance of this state machine, which is identified through the BindID in exchanged PDUs and API interactions. Because most operations of GAP are processed by the GSA synchronously, there is a number of states which are treated identically, which is they are entered by issuing a request, which causes the GUA to send a PDU to the GSA, and left by receiving the response, which generates a confirmation. The states treated in this way are bind application pending, unbind application pending, create pending, modify pending, join group pending, leave session pending, leave group pending, delete pending, renegotiate pending, and invite pending.

Because the GSA may send some PDUs without any trigger from the GUA, these events must be handled in all states of the user bound compound state. The events falling in this category are ManagerRQ (causing a ManagerInd), NotificationRQ (causing a NotificationInd), InvitationRQ (causing an InvitationInd), and RenegotiationRQ (causing a RenegotiationInd). Because it must also be possible for a user to always respond to a ManagerInd with a ManagerRes without having to wait for another

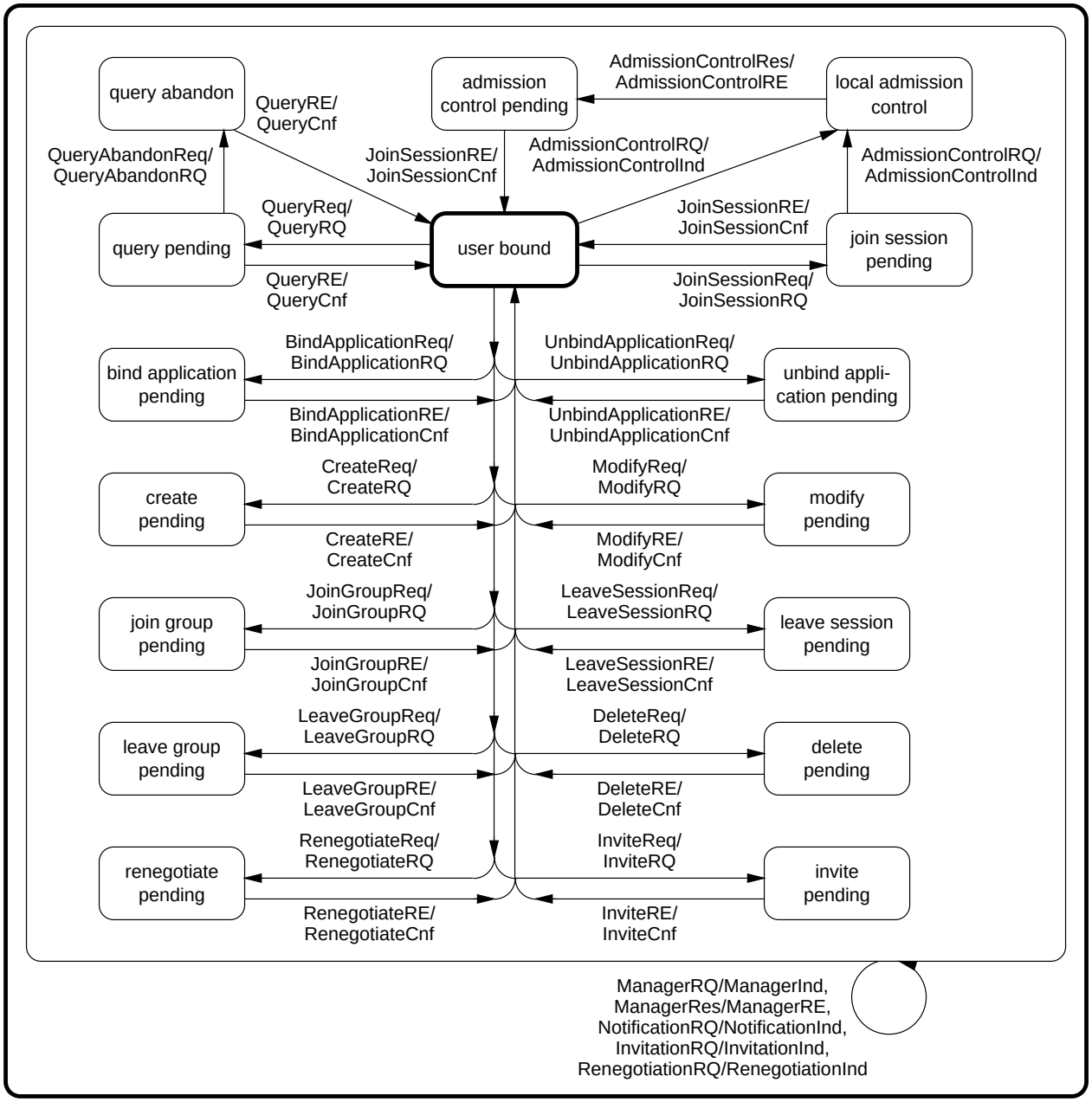

Figure 8: GAP/GUA state diagram (user bound compound state)

operation to finish, ManagerRes may also be issued in all states of the user bound compound state and will always cause the GUA to send a ManagerRE PDU to the GSA. All events and actions described in this paragraph do not change the GUA'a state.

Two operations which may take longer to complete are query and join session. Because of this difference, they are treated differently in the GUA's state machine. A query may be issued with a QueryReq, which causes the GUA to send a QueryRQ PDU to the GSA and go into the query pending state. If the query has been completely processed by the GSA, it responds with a QueryRE PDU which causes the GUA to generate a QueryCnf and to go back to the user bound state. Another way to leave the query pending state is to issue a QueryAbandonReq, which will cause the GUA to move to the query abandon state and to send a QueryAbandonRQ to the GSA. The GUA then waits for a QueryRE PDU confirming the query abandon, which will generate a QueryCnf and make the transition back to the query pending state.

Joining a session is initiated by issuing a JoinSessionReq, which will cause the GUA to move to the join session pending state and send a JoinSessionRQ PDU to the GSA. Depending on how `waitForManagerReplies` is set in the JoinSessionRQ PDU (described in section 3.2.11), the GSA behaves differently. If `asynchronousTimeout` is selected, then the GSA responds with a JoinSessionRE PDU containing a `pendingJoinId`, which is used to refer to the JoinSessionRQ. The GUA will then make the transition back to the user bound state and generate a JoinSessionCnf. If the `synchronousTimeout` attribute is selected, the GSA will wait for enough manager replies and then either respond with a JoinSessionRE- PDU which will generate a JoinSessionCnf- and take the GUA back to the user bound state (if the join session has been rejected by the session managers), or respond with an AdmissionControlRQ PDU containing the information necessary for joining the session, which will generate an AdmissionControlInd and make the transition to the local admission control state (if the session managers accepted the join session request). The transition to the local admission control state is also possible from the user bound state, if the user selected `asynchronousTimeout`, the session managers accepted the join session request and the GSA sends an AdmissionControlRQ PDU (containing the `pendingJoinId` to refer to the JoinSessionRQ), which will generate an AdmissionControlInd. Being in the local admission control state, the user must perform a local admission control, deciding on which flows to join and whether the local resources are sufficient for actually joining them. The result of the local admission control is issued in an AdmissionControlRes, which will cause the GUA to send a AdmissionControlRE PDU to the GSA and make the transition to the admission control pending state. Only if the GSA confirms the AdmissionControlRE PDU with a JoinSessionRE+, the join may actually be performed. Otherwise, the join has to be cancelled. The JoinSessionRE PDU received from the GSA will generate a JoinSessionCnf and the transition back to the user bound state.

3.2 PDU definitions

The PDUs exchanged by the communicating entities (GUA and GSA) 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 [6].

The following module heading for the definition of the GAP PDUs first lists all the necessary imports from the modules described earlier in this specification and then defines some identification types to be integers. The definitions of `GuaPdu` and `GsaPdu` are necessary for decoding incoming PDUs properly. They are simply a list of the PDUs being sent by the GUA or GSA, respectively.

```
GAP-PDUS DEFINITIONS ::= BEGIN

IMPORTS
    Application, AuthType, 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;

BindId ::= INTEGER

ManagerRequestId ::= INTEGER

PendingJoinId ::= INTEGER

TmpBindId ::= INTEGER

GuaPdu ::= --snacc isPdu:"TRUE" -- CHOICE {
    bindGuaRequest              [0]     BindGuaRequest,
    bindUserRequest             [1]     BindUserRequest,
    authenticateResponse        [2]     AuthenticateResponse,
    challengeResponse           [3]     ChallengeResponse,
    bindApplicationRequest      [4]     BindApplicationRequest,
    unbindApplicationRequest    [5]     UnbindApplicationRequest,
    unbindUserRequest           [6]     UnbindUserRequest,
    unbindGuaRequest            [7]     UnbindGuaRequest,
    createRequest               [8]     CreateRequest,
    queryRequest                [9]     QueryRequest,
    queryAbandonRequest         [10]    QueryAbandonRequest,
    modifyRequest               [11]    ModifyRequest,
    joinGroupRequest            [12]    JoinGroupRequest,
    joinSessionRequest          [13]    JoinSessionRequest,
    admissionControlResponse    [14]    AdmissionControlResponse,
    leaveSessionRequest         [15]    LeaveSessionRequest,
    leaveGroupRequest           [16]    LeaveGroupRequest,
    deleteRequest               [17]    DeleteRequest,
    renegotiateRequest          [18]    RenegotiateRequest,
    inviteRequest               [19]    InviteRequest,
    managerResponse             [20]    ManagerResponse }

GsaPdu ::= --snacc isPdu:"TRUE" -- CHOICE {
    bindGuaResponse             [0]     BindGuaResponse,
    authenticateRequest         [1]     AuthenticateRequest,
    challengeRequest            [2]     ChallengeRequest,
    bindUserResponse            [3]     BindUserResponse,
    bindApplicationResponse     [4]     BindApplicationResponse,
    unbindApplicationResponse   [5]     UnbindApplicationResponse,
    unbindUserResponse          [6]     UnbindUserResponse,
    unbindGuaResponse           [7]     UnbindGuaResponse,
    createResponse              [8]     CreateResponse,
    queryResponse               [9]     QueryResponse,
    modifyResponse              [10]    ModifyResponse,
    joinGroupResponse           [11]    JoinGroupResponse,
    joinSessionResponse         [12]    JoinSessionResponse,
```

```
    admissionControlRequest     [13]    AdmissionControlRequest,
    leaveSessionResponse        [14]    LeaveSessionResponse,
    leaveGroupResponse          [15]    LeaveGroupResponse,
    deleteResponse              [16]    DeleteResponse,
    renegotiateResponse         [17]    RenegotiateResponse,
    inviteResponse              [18]    InviteResponse,
    managerRequest              [19]    ManagerRequest,
    notificationRequest         [20]    NotificationRequest,
    invitationRequest           [21]    InvitationRequest,
    renegotiationRequest        [22]    RenegotiationRequest }
```

The individual PDUs will be described in detail in the following subsections. For each GAP service (which usually consists of multiple PDUs being exchanged according to the rules defined in section 3.1), a description of the service in general is given, followed by the individual PDUs and a detailed description of each PDU and its fields.

3.2.1 Bind GUA

The bind GUA service is used by a GUA to connect to a GSA and to make sure that communication between both components is possible. As specified in the state diagrams in section 3.1, the bind GUA service is the first service to use in the communication between a GUA and a GSA.

```
BindGuaRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    gapVersion                  [0]     SEQUENCE {
        majorRelease                [0]     INTEGER,
        minorRelease                [1]     INTEGER,
        releaseInformation          [2]     IA5String },
    guaDescription              [1]     IA5String OPTIONAL }
```

The `BindGuaRequest` PDU sent from the binding GUA to the GSA contains information about the protocol version the GUA implements. This information is given in the `gapVersion` attribute. This attribute contains a `majorRelease` number, a `minorRelease` number, and an optional `releaseInformation` string, which can be used to specify a comment characterizing the GAP version. An optional `guaDescription` string can be used for a short description of the binding GUA.

```
BindGuaResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindGuaResult               [0]     ENUMERATED {
        success                     (1),
        noMoreGuas                  (2),
        versionMismatch             (3) },
    gapVersion                  [1]     SEQUENCE {
        majorRelease                [0]     INTEGER,
        minorRelease                [1]     INTEGER },
    maxNumUsers                 [2]     INTEGER OPTIONAL,
    gsaDomain                   [3]     GmsDomainName OPTIONAL }
```

The `BindGuaResponse`, which is sent back by the GSA as a response to a `BindGuaRequest`, contains information about the success of the bind GUA service request. Two error cases, which are specified in the `bindGuaResult` attribute are `noMoreGuas` and `versionMismatch`. In case of the `noMoreGuas` error, the GSA can not accept a new GUA since there are no more resources to handle one more GUA. However, if the same bind GUA request is issued after another GUA has unbound from the GSA, the bind GUA request would be accepted. In case of the `versionMismatch` error, the GSA replies with the message that the GAP version defined by the GUA in the `BindGuaRequest` PDU can not be handled correctly by the GSA. It therefore rejects the bind GUA request.

In case of a `bindGuaResult` with the value `success`, the GSA accepts the bind GUA request and responds with information about the binding. The first information (which is also given in case of rejected bind GUA requests described in the previous paragraph), the `gapVersion` implemented by the GSA is given as a `majorRelease` number and a `minorRelease` number. This version is compatible with the version requested by the GUA (otherwise, the GSA responds with a `bindGuaResult` of `versionMismatch`). The next information is optional and defines the `maxNumUsers`, ie the maximum number of users which can be bound concurrently using this GUA binding. Depending on the GSA's implementation, this number is known in advance (and given in the `BindGuaResponse`), or it is decided dynamically, whether a new user can be accepted (in which case the `maxNumUsers` attribute is not given). The `gsaDomain` attribute gives information about the domain the GSA is in.

3.2.2 Bind User

The bind user service is used by a user to bind to the GMS. Because one of the goals of GMS is to provide identification, authentication, and authorization to applications using it, it is not possible to use any GMS services related to users, groups, and sessions without first binding to the GMS. The bind user service is the most complicated service of GMS, and the description of PDUs is much easier to understand when figures 4 and 7 are used as references describing the possible exchanges of PDUs.

```
BindUserRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    tmpBindId                   [0]     TmpBindId,
    userIdentification          [1]     CHOICE {
        anonymous                   [0]     NULL,
        userName                    [1]     GmsObjectName },
    transportServices           [2]     SET OF TransportService }
```

The `BindUserRequest` PDU is sent from a GUA to a GSA. Because this is the first PDU associated with information about a specific user (in contrast to the bind GUA service), a `tmpBindId` must be supplied which identifies the bind user request for the duration of the bind user service. The GUA may choose any value for the `tmpBindId`, provided it is not used in another bind user service procedure running in parallel. If the bind user service is successful, the user binding will be identified by a `bindId` for the remainder of the binding's lifetime, which is selected by the GSA and transmitted to the GUA in the bind user response.

The second parameter of the `BindUserRequest` PDU is the `userIdentification`, which is used to identify the user who is using the bind user service. If a user wants to bind as `anonymous`, it is not necessary to submit an identification. If a user wants to bind with his `userName` (which will be the normal case, because authorization is based on the identity of the user), the name must be supplied.

The third parameter of the `BindUserRequest` PDU is the `transportServices` attribute which is used to identify the transport infrastructure being used by the user. Normally, this would only be one value, identifying the transport infrastructure of which the GUA is part of. However, it is possible that one GUA is used for several transport infrastructures.

```
AuthenticateRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    tmpBindId                   [0]     TmpBindId,
    authenticationMethods       [1]     SET OF SET {
        authType                    [0]     AuthType,
        numOfChallengesRequired     [1]     INTEGER OPTIONAL } }
```

If the bind user service did not specify a non-existing user name or the request to bind anonymously, the GSA will send back a `AuthenticateRequest` which contains information about the possible ways for the user to bin to the GMS. The `tmpBindId` identifies the user who is addressed.

The authenticationMethods attribute is a set of available authentication methods for the requesting user at the GSA he is using. This set is an intersection of the user's authInformation (described in section 2.3.1) and the authentication methods supported by the GSA. Each entry of this set contains the authType (defined in section 2.1) and an optional numOfChallengesRequired, which specified how many challenges a user has to answer in order to use this authentication method.

```
AuthenticateResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    tmpBindId                   [0]     TmpBindId,
    authenticateResult          [1]     ENUMERATED {
        authentifying               (1),
        abortAuthentication         (2) },
    authenticationMethod        [2]     AuthType OPTIONAL,
    authData                    [3]     OCTET STRING OPTIONAL }
```

The AuthenticateResponse PDU is sent from the GUA to the GSA. It contains a tmpBindId attribute which is used to identify the user for whom the response is given. The authenticateResult specifies whether the user wants to continue the authentication process (in which case the authentifying value is used) or whether the user decided to cancel the authentication process (in which case the abortAuthentication value is used). In the second case, the two optional attributes of the AuthenticateResponse PDU are nor present.

If the authentifying value is used, the authenticationMethod attribute has to be present, identifying the authentication method the user wants to use. If this method requires the submission of additional data, the authData attribute also is used, containing the authentication data. In case of the unixPassword authentication type, this attribute would contain the password of the user who wants to bind to the GMS.

```
ChallengeRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    tmpBindId                   [0]     TmpBindId,
    challengeResult             [1]     ENUMERATED {
        success                     (1),
        failed                      (2) },
    challengeData               [2]     SEQUENCE OF OCTET STRING,
    remainingChallenges         [3]     CHOICE {
        last                        [0]     NULL,
        unknown                     [1]     NULL,
        numberOfChallenges          [2]     INTEGER } }
```

If the authentication method selected by the user in the AuthenticateResponse PDU requires challenges, the GSA sends a ChallengeRequest PDU to the GUA. This PDU is also used if the user is already in the challenging state and more challenges have to be answered for the authentication. It contains a tmpBindId attribute which is used to identify the request is sent. The challengeResult contains the result of the previous challenge (if this challenge request is the first one, this attribute is always set to success). The challengeData attribute contains a sequence of challenges. In most cases, only one challenge will be sent in each ChallengeRequest PDU.

The remainingChallenges attribute contains information about the status of the challenging progress. If the challenge request is the last one, the attribute has the value last. If the number of following challenge requests is not known, the attribute has the value unknown. If the number of challenges is known, it is given as an integer value in the numberOfChallenges attribute.

```
ChallengeResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    tmpBindId                   [0]     TmpBindId,
    challengeResult             [1]     ENUMERATED {
        challenging                 (1),
```

```
        abortChallenge              (2) },
    challengeResponseData       [2]     SEQUENCE OF OCTET STRING OPTIONAL }
```

The `ChallengeResponse` PDU is sent from the GUA to the GSA. It contains a `tmpBindId` attribute which is used to identify the user for whom the response is given. The `challengeResult` specifies whether the user wants to continue the challenging process (in which case the `challenging` value is used) or whether the user decided to cancel the challenging process (in which case the `abortChallenge` value is used). In the second case, the optional `challengeResponseData` contains the responses to the challenges sent in the last challenge request.

```
BindUserResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    tmpBindId                   [0]     TmpBindId,
    bindUserResult              [1]     ENUMERATED {
        success                     (1),
        noSuchUser                  (2),
        noMoreUsers                 (3),
        authFailed                  (4),
        authenticationAborted       (5),
        challengingAborted          (6) },
    bindId                      [2]     BindId OPTIONAL }
```

The `BindUserResponse` PDU is sent from the GSA to the GUA. It contains a `tmpBindId` attribute which is used to identify the user for whom the response is given. The second attribute is `bindUserResult`, which specifies the result of the bind user service. If the `success` value is present, the bind user service completed successfully and the user is bound to the GMS. In this case, the optional third argument of the `BindUserResponse` PDU contains a `bindId` value which will to be used in all future services which address this user.

The `noSuchUser` result is given as an immediate answer to a bind user request and specifies that the user's name who was given in the request does not refer to an existing user. If the GSA can not handle more users, the `noMoreUsers` value will be returned. If the authentication process fails because the user did not supply the correct data in an authentication or challenge response, the `bindUserResult` will be set to `authFailed`. If the user selected to abort the authentication, the `authenticationAborted` value will be returned. If the user selected to abort the challenging, the `challengingAborted` value will be returned.

3.2.3 Bind Application

The 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 normal case will be only one application per user binding, but GMS is not limited to this model. Applications are used within user bindings (described in section 2.3.1) and sessions (described in section 2.3.5). Currently, GMS does not enforce a user to bind a application (ie a user may join a session although he has not bound the application which is registered in the session's `sessionAppInfo`). However, future GAP versions will include a stronger approach to the relationship between applications and sessions.

```
BindApplicationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    application                 [1]     Application }
```

The `BindApplicationRequest` PDU is sent from a GUA to the GSA. It includes a `bindId` attribute which identifies the user requesting the bind application service. The second attribute is the `application` identification, which is described in section 2.1. It specifies the application which shall be bound.

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

The `BindApplicationResponse` PDU is sent back from the GSA to the GUA. It contains a `bindId` attribute which is used to identify the user for whom the response is given. The `bindApplicationResult` specifies the result of the bind application request. A value of `success` indicates the success of the bind application request. In this case, the application has been added to the user's current `bindings` as described in section 2.3.1. In case of a `bindApplicationResult` attribute value of `applicationAlreadyBound`, the application has already been bound for this binding.

3.2.4 Unbind Application

The unbind application service is used by a user to notify the GMS that an application which has been bound using the bind application service (described in section 3.2.3) is now unbound. This normally means that the application terminated.

```
UnbindApplicationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    application                 [1]     Application }
```

The `UnbindApplicationRequest` PDU is sent from a GUA to the GSA. It includes a `bindId` attribute which identifies the user requesting the unbind application service. The second attribute is the `application` identification, which is described in section 2.1. It specifies the application which shall be unbound.

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

The `UnbindApplicationResponse` PDU is sent back from the GSA to the GUA. It contains a `bindId` attribute which is used to identify the user for whom the response is given. The `bindApplicationResult` specifies the result of the unbind application request. A value of `success` indicates the success of the unbind application request. In this case, the application has been removed from the user's current `bindings` as described in section 2.3.1. In case of a `unbindApplicationResult` attribute value of `noSuchApplication`, the application which has been specified in the `application` attribute of the `UnbindApplicationRequest` PDU is not part of the user's current `bindings`.

3.2.5 Unbind User

The unbind user service is used by a user to unbind from the GMS. The unbind user service is symmetric to the bind user service described in session 3.2.2, which is used to bind a user to the GMS. Because it is possible that a user did not properly leave all sessions before using the unbind user service, it is possible to specify what to do with open sessions.

```
UnbindUserRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    forceUnbind                 [1]     BOOLEAN }
```

The `UnbindUserRequest` PDU is sent from a GUA to a GSA. It first contains a `bindId` which identifies the user who is issuing the unbind user request. The `forceUnbind` attribute specifies what to do with open sessions. If it is set to true, the unbind user will cause the user to leave all sessions he is a participant of within this user binding. If the `forceUnbind` attribute set to false, the unbind user service will only succeed if the user has left all open sessions.

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

The `UnbindUserResponse` PDU is sent back from a GSA to a GUA. The `bindId` identifies the user who requested the unbind user service. The `unbindUserResult` attribute contains the result of the unbind user service. A value of `success` indicates that the unbind user service was successful and the user has been unbound. This has also happened if a result of `terminatedSessions` is returned, but in this case there were open sessions which were left. In both cases, the `bindId` refers no longer to an existing user binding and can therefore not be use anymore.

If an `unbindUserResult` of `activeSessions` is returned, there were open sessions which were not left because the `forceUnbind` attribute in the `UnbindUserRequest` had been set to false. In this case, the list of open sessions is returned in the optional `activeSessions` attribute. The user may then leave all these sessions explicitly (using the leave session service described in section 3.2.12) or use the unbind user service and set the `forceUnbind` attribute to true.

3.2.6 Unbind GUA

The unbind GUA service is used for disconnecting a GUA from a GSA. The unbind GUA service is symmetric to the bind GUA service described in session 3.2.1, which is used to connect a GUA to a GSA. When using the unbind GUA service, it is necessary to specify how active user bindings should be handled. In addition to active users (ie users which have bound to the GSA using the bind user service described in section 3.2.2), it is also possible that these users are active within sessions (ie they joined a session using the join session service described in section 3.2.11). Because of these possibilities, the GUA must specify what to do in these cases.

```
UnbindGuaRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    forceUnbind                 [0]     ENUMERATED {
        noForce                         (1),
        forceUserUnbinds                (2),
        forceSessionLeaves              (3) } }
```

The `UnbindGuaRequest` PDU, which is sent by the GUA wishing to disconnect to the GSA it is currently bound to, only contain information about the `forceUnbind` mode, which specifies what to do with bound users and active sessions. If this attribute is set to `noForce`, then the unbind GUA request only succeeds if there is no user bound to the GMS using this GUA. If it is set to `forceUserUnbinds`, then users being bound to the GMS (ie they have used the bind user service successfully and did not use the unbind user service successfully afterwards) will be unbound automatically, but only if they are not active in a session. If the `noForce` attribute is set to `forceSessionLeaves`, then also users being active in a session (ie they used the join session service successfully and did not use the leave session service successfully afterwards) will be unbound automatically. This last variant of the unbind GUA service will always be successful.

```
UnbindGuaResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    unbindGuaResult             [0]     ENUMERATED {
        success                         (1),
        terminatedUserBindings          (2),
        terminatedSessions              (3),
        activeUsers                     (4),
        activeUsersActiveSessions       (5) },
    activeUsers                 [1]     SET OF SET {
        bindId                          [0]     BindId,
        userName                        [1]     GmsObjectName,
        sessions                        [2]     SET OF GmsObjectName OPTIONAL } OPTIONAL }
```

The `UnbindGuaResponse` PDU is sent back by the GSA to a GUA as a response to a `UnbindGuaRequest` PDU this GUA sent previously. Depending on the `forceUnbind` attribute in the `UnbindGuaRequest` PDU, the `unbindGuaResult` can have several values. In case of `success`, the unbind GUA service request was successful and it was not necessary to terminate any user binding or user being active in a session. The GUA can close the connection to the GSA after it received this response.

The `unbindGuaResult` values of `terminatedUserBindings` and `terminatedSessions` indicate that it was necessary to terminate user bindings or both user bindings and users being active in a session. In case of these `unbindGuaResult` values, the list of the affected users (and, if present, the respective sessions), is contained in the `activeUsers` attribute. It is up to the GUA to signal the termination of the GAP connection to the users. The GSA will act as if the users left the sessions and issued unbind user requests. The GUA can close the connection to the GSA after it received one of these responses.

The `unbindGuaResult` values of `activeUsers` and `activeUsersActiveSessions` indicate that the value given in the `forceUnbind` attribute of the `UnbindGuaRequest` PDU was not strong enough to force the unbind GUA request to succeed. The list of users and, in case of `activeUsersActiveSessions`, sessions, which prevented the unbind GUA request to succeed is given in the `UnbindGuaResponse` PDU in the `activeUsers` attribute. It is up to the GUA to decide whether it will force a unbind GUA with a stronger `forceUnbind` attribute, or whether it will notify the users given in the `activeUsers` attribute in order to give them the chance to leave the sessions and unbind themselves without being forced to do so. Consequently, the GUA can not close the connection to the GSA after it received one of these responses.

3.2.7 Create

The create service is used to create GMS objects. Creating GMS objects is possible for all users (except anonymously bound users which are described in section 2.3.1), but this will change in future versions of GAP. Currently, every user is allowed to created one of the objects described in section 2.3, and the name of the created object is not restricted (ie the object to be created may be in any existing domain). The only exception from this rule is the flow object type. Objects of this type are created (and deleted) automatically when a session is created (or deleted), so it is not possible to create only a flow object with the create service.

```
CreateRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    objectName                  [1]     GmsObjectName,
    objectType                  [2]     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,
        flowAddressingInformation   [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 a GUA to a GSA. It first contains a `bindId` which identifies the user who is issuing the create service request. The `objectName` attribute gives the name of the object which is to be created. This name is a full `GmsObjectName` as defined in section‘2.1, so any domain name can be given. This way it is possible to create an object in any domain, even if the GSA the user is bound to is not part of this domain. 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. 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`. 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.

```
CreateResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    createResult                [1]     ENUMERATED {
        success                         (1),
        noPermission                    (2),
        nameInUse                       (3),
        nonExistingManager              (4),
        noSuchGroup                     (5),
        noSuchUser                      (6),
        alreadyIndirectMember           (7),
        cyclicFlowDependencies          (8) } }
```

The `CreateResponse` PDU is sent back from the GSA to the GUA. The `bindId` identifies the user who requested the create service. The only other attribute of the `CreateResponse` PDU is the `createResult`. A value of `success` indicates that the create service request was processed successfully and the requested object has been created.

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.

3.2.8 Query

The query service is used to query the GMS about objects. In general, there are two query modes available. When using the search mode, the 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 GAP. In this version of GAP, only perfect matches to attribute values are possible. However, if a attribute values are empty or set to zero, GMS will not try to match this value but only use the other attribute values for finding matching objects. The result of searching for objects is a list of object names.

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.

```
QueryRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    queryType                   [1]     CHOICE {
        search                      [0]     SET {
            queryTemplate               [0]     CHOICE {
                object                      [0]     GmsObject,
                relation                    [1]     GmsRelation },
            queryScope                  [1]     CHOICE {
                localDomain                 [0]     NULL,
                domainList                  [1]     SET OF GmsDomainName,
                global                      [2]     NULL },
            maxNumberOfMatches          [2]     INTEGER },
        get                         [1]     CHOICE {
            object                      [0]     GmsObjectName,
            relation                    [1]     GmsRelationName } } }
```

The `QueryRequest` PDU is sent by a GUA to a GSA. It first contains a `bindId` which identifies the user who is issuing the query service request. 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 found 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 an `relation`, the `relationName` will not be interpreted.

The `queryScope` attribute determines the scope of the search. It can have one of three values. When setting the `queryScope` attribute to `localDomain` only the local domain will be used for the search. If other domains than the local one should be searched, it is possible to specify a `domainList` which is a set of domains to be searched. The third value for the `queryScope` attribute is `global` which specifies that the search should be performed globally. 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 user can select whether he wants to get an `object` or a `relation`. In both cases, the name must be specified. 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 {
    bindId                      [0]     BindId,
    abandonType                 [1]     ENUMERATED {
        dismissResults              (1),
        sendPartialResults          (2) } }
```

Because the query service may take a long time to continue, it is possible to abandon a running query request with the `QueryAbandonRequest` PDU which is sent from the GUA to the GSA. In figures 8 and 5 it can be seen that this PDU may be sent by the GUA at any after sending a `QueryRequest` PDU and before receiving a `QueryResponse` PDU. The `QueryAbandonRequest` PDU has only two attributes, the first one being the `bindId` to identify the user whose query service shall be abandoned. 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 user will not receive any results in the `QueryResponse` PDU. 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 the `QueryResponse` PDU.

```
QueryResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    queryResult                 [1]     ENUMERATED {
        success                         (1),
        abandonedNoResults              (2),
        abandonedPartialResults         (3),
        permissionDenied                (4),
        tooManyMatchesRequired          (5) },
    queryMatches                [2]     CHOICE {
        search                          [0]     CHOICE {
            objects                             [1]     SET OF GmsObjectName,
            relations                           [0]     SET OF GmsRelationName },
        get                             [1]     CHOICE {
            object                              [0]     GmsObject,
            relation                            [1]     GmsRelation } } OPTIONAL }
```

The `QueryResponse` PDU is sent from the GSA to the GUA. It is either a response to a `QueryRequest` PDU (if the query service completed normally) or to a `QueryAbandonRequest` (if the query service was interrupted). It contains a `bindId` attribute which is used to identify the user for whom the response is given. The `queryResult` specifies the result of the join group 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.

Two possible results are errors responses from the GSA. 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.

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.

3.2.9 Modify

The 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 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.

```
ModifyRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    modifyType                  [1]     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 s sent by a GUA to a GSA. It first contains a `bindId` which identifies the user who is issuing the modify service request. The `modifyType` attribute identifies the modify mode which is selected. When using the `object` mode, it is possible to modify an object. The `objectName` attribute identifies the object, 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 GAP.

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 {
    bindId                      [0]     BindId,
    modifyResult                [1]     ENUMERATED {
        success                         (1),
        noSuchObject                    (2),
        noSuchRelation                  (3),
        noPermission                    (4),
        lastManager                     (5),
        lastOwner                       (6) } }
```

The ModifyResponse PDU is sent from the GSA to the GUA. It contains a bindId attribute which is used to identify the user for whom the response is given. 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.

3.2.10 Join Group

The 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 use the join group service asynchronously, ie a user is not blocked until the join group request is completely processed.

```
JoinGroupRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    groupName                   [1]     GmsObjectName,
    joiningGroupName            [2]     GmsObjectName OPTIONAL,
    waitForManagerReplies       [3]     CHOICE {
        synchronousTimeout          [0]     INTEGER,
        asynchronousTimeout         [1]     INTEGER } }
```

The JoinGroupRequest PDU is sent by a GUA to a GSA. It first contains a bindId which identifies the user who is issuing the join group request. The groupName attribute specifies the group's name the user wants to join. If the optional joiningGroupName attribute is present, it is not the user who want to join the group, but the group specified by this argument shall join the group specified by the groupName attribute. Joining groups to groups is only allowed if the user is manager of both groups. The waitForManagerReplies argument specifies how the request is to be processed by the GSA.

If the waitForManagerReplies is set to synchronousTimeout, the GSA will try process the join group request (which may involve a number of manager requests, depending on the group's groupJoinPolicy) for at most the amount of time in seconds specified by this attribute. If the synchronousTimeout attribute is set to zero, the GSA will not use a timeout at all. This setting may be dangerous, because the user is blocked in the join group pending state (according to the state diagram shown in figure 8). The user then entirely depends on the managers to reply to the manager requests (described in section 3.2.17).

If the waitForManagerReplies is set to asynchronousTimeout, the GSA will respond immediately with a pendingJoinId and process the join request asynchronously to the user's activities. Consequently, the user is not blocked and may use other services while the join request is being processed. It will be processed for at most the amount of time in seconds specified by the asynchronousTimeout attribute. If the attribute is set to zero, the GSA will not use a timeout at all. The result of the asynchronously processed join group request will be sent to the requesting user as a notification as described in section 3.2.18.

```
JoinGroupResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    joinGroupResult             [1]     ENUMERATED {
        success                         (1),
        timeout                         (2),
        refuse                          (3),
        noSuchGroup                     (4),
        authenticationInsufficient      (5),
        notAuthorized                   (6),
        alreadyMember                   (7),
        alreadyIndirectMember           (8),
        joinPending                     (9),
        joinPendingLimitExceeded        (10) },
    pendingJoinId               [2]     PendingJoinId OPTIONAL }
```

The JoinGroupResponse PDU is sent back by the GSA to the GUA. It contains a bindId attribute which is used to identify the user for whom the response is given. The joinGroupResult specifies the result of the join group request. A value of success indicates the success of the join group request. It indicates that the join group request was successfully completed and the requested join has been performed. Consequently, the user or the group (if the joiningGroupName has been specified in the JoinGroupRequest PDU) now is a new member of the group (if the group's groupNotificaPolicy is set accordingly, this will be notified to all members and/or managers of the group).

A joinGroupResult attribute value of timeout will only occur if the join group request's waitForManagerReplies attribute has been set to a synchronousTimeout value greater than zero 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 waitForManagerReplies attribute has been set to synchronousTimeout.

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 binding's AuthType described in section 2.3.1). 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.

To 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. In this case, the `JoinGroupResponse` will contain the optional `pendingJoinId`, which will be used in a future notification by the GSA (described in section 3.2.18) to send the eventual result of the join group request to the GUA. 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.

3.2.11 Join Session

The 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 use the join session service asynchronously, ie a user is not blocked until the join session request is completely processed.

```
JoinSessionRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    sessionName                 [1]     GmsObjectName,
    flows                       [2]     SET OF SET {
        flowName                    [0]     GmsObjectName,
        joinRole                    [1]     ENUMERATED {
            sender                          (1),
            receiver                        (2),
            senderAndReceiver               (3) } },
    waitForManagerReplies       [3]     CHOICE {
        synchronousTimeout          [0]     INTEGER,
        asynchronousTimeout         [1]     INTEGER } }
```

The `JoinSessionRequest` PDU is sent by a GUA to a GSA. It first contains a `bindId` attribute which identifies the user who is issuing the join session request. The `sessionName` attribute specifies the session's name 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 `waitForManagerReplies` argument specifies how the request is to be processed by the GSA.

If the `waitForManagerReplies` is set to `synchronousTimeout`, the GSA will try process the join session request (which may involve a number of manager requests, depending on the group's `sessionJoinPolicy`) for at most the amount of time in seconds specified by this attribute. If the `synchronousTimeout` attribute is set to zero, the GSA will not use a timeout at all. This setting may be dangerous, because the user is blocked in the `join session pending` state (according to the state diagram shown in figure 8). The user then entirely depends on the managers to reply to the manager requests (described in section 3.2.17).

If the `waitForManagerReplies` is set to `asynchronousTimeout`, the GSA will respond immediately with a `pendingJoinId` and process the join request asynchronously to the user's activities. Consequently, the user is not blocked and may use other services while the join session request is being processed. It will be processed for at most the amount of time in seconds specified by the `asynchronousTimeout` attribute. If the attribute is set to zero, the GSA will not use a timeout

at all. The result of the asynchronously processed join session request will either be sent to the requesting user as a notification as described in section 3.2.18 (if the result is refuse or timeout), or it will be sent in an admission control request, as described in this session (if the authorization control was successful).

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

The `JoinSessionResponse` PDU is sent back from the GSA to the GUA. Depending on the requested mode (as specified in `waitForManagerReplies`) and the result of the authorization control, the join session response will be preceded by an `AdmissionControlRequest` PDU. The possible ways how the join session service may be performed are described in section 3.1. The `bindId` attribute is used to identify the user for whom the response is given.

The `joinSessionResult` specifies the result of the join session request. A value of `success` indicates the success of the join session request. It indicates that the join session request was successfully completed and the requested join has been performed. Consequently, the user now is a new participant of the session (if the session's `sessionNotifiPolicy` is set accordingly, this will be notified to all participants and/or managers of the session). However, if the `JoinSessionResponse` PDU with the `joinSessionResult` attribute set to `success` is received after the GUA sent a admission control response with a `admissionControlResult` attribute value of `admissionControlFailed`, it does not mean that the join was successful, but that the join session request was successfully cancelled because of scarce resources at the GUA side.

A `joinSessionResult` attribute value of `timeout` will only occur if the join session request's `waitForManagerReplies` attribute has been set to a `synchronousTimeout` value greater than zero 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 to `synchronousTimeout`.

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. In this case, the `JoinSessionResponse` will contain the optional `pendingJoinId`, which will be used in a future notification or admission control request by the GSA (depending on the result of the manager queries) to send the eventual result of the join session request to the GUA. If the GSA is not able to handle more asynchronous join session requests, it will respond with a `joinPendingLimitExceeded` result. It is then up to the user to decide whether to process the join session request synchronously or to wait some time and to try again with an asynchronous join session request.

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 and the GUA has to perform a roll-back.

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.

In general, in case of the `JoinSessionResponse` PDU it is important to notice that this PDU may be sent one or two times, depending on the `waitForManagerReplies` attribute in the request and the result of the authorization control. If the `waitForManagerReplies` attribute is set to `synchronousTimeout`, the `JoinSessionResponse` PDU will only be sent once (either directly after the `JoinSessionRequest` PDU or after exchanging admission control PDUs). If the `waitForManagerReplies` attribute is set to `asynchronousTimeout`, the `JoinSessionResponse` PDU is sent back after a parameter check. If the authorization control yields a timeout or a refuse result, this will be notified with a notification service PDU (described in section 3.2.18). If the authorization control is successful, admission control PDUs will be exchanged followed by a second `JoinSessionResponse` PDU. Section 3.1 contains a graphical representation of these procedures.

```
AdmissionControlRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    pendingJoinId               [1]     PendingJoinId OPTIONAL,
    flows                       [2]     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), a `AdmissionControlRequest` PDU is sent from the GSA to the GUA. This happens regardless of synchronous or asynchronous mode of the join session service. The `bindId` identifies the

user who is receiving the admission control request. In case of asynchronous operation, the optional `pendingJoinId` attribute identifies the join session request to which this admission control request belongs.

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.

Upon receiving the `AdmissionControlRequest` PDU, the user (or the communication framework used by the user as described in section 1) must perform a local admission control. The purpose of the admission control is to check whether the resources available are sufficient to join all flows with the QoS values specified in the flows' specifications. It is possible to not join some of the flows specified in the `flows` attribute of the `JoinSessionRequest` PDU, as long as all existing relations are satisfied.

```
AdmissionControlResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    admissionControlResult      [1]     ENUMERATED {
        success                         (1),
        admissionControlFailed          (2) },
    flowsBeingJoined            [2]     SET OF SET {
        flow                            [0]     GmsObjectName,
        receiverAddress                 [1]     OCTET STRING OPTIONAL } OPTIONAL }
```

The `AdmissionControlResponse` PDU is sent back from the GUA to the GSA. It contains the result of the admission control. The first attribute is the `bindId` which identifies the user who is responding. The second attribute of the `AdmissionControlResponse` PDU is the `admissionControlResult`, which specifies the result of the local admission control. If this attribute is set to `success`, it indicates that the local admission control was successful, ie the local resources were sufficient to perform the join session operation. If the `admissionControlResult` is set to `admissionControlFailed`, the local admission control failed (ie there were not enough resources available) and the join session service is cancelled, ie the user will not become a participant of the session.

The `flowsBeingJoined` attribute of the `AdmissionControlResponse` PDU specifies which flows are eventually being joined by the user. This optional attribute is only present if the local admission control was successful. 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) 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.

In general, the join session service is only successfully completed if the GSA replies with a `JoinSessionResponse` PDU containing a `joinSessionResult` attribute with a value of `success`. If the result is `refuse` or `missingDependencies`, the GSA decided to cancel the join session service and the resources reserved by the GUA during the admission control process can be released.

3.2.12 Leave Session

The 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.

```
LeaveSessionRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    sessionName                 [1]     GmsObjectName }
```

The `LeaveSessionRequest` PDU is sent from a GUA to a GSA. The `bindId` attribute identifies the user who is requesting the leave session service. The `sessionName` attribute specifies the session to be left.

```
LeaveSessionResponse  ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    leaveSessionResult          [1]     ENUMERATED {
        success                         (1),
        noSuchSession                   (2) } }
```

The `LeaveSessionResponse` PDU is sent back from the GSA to the GUA. The `bindId` identifies the user who requested the leave session service. The only other attribute of the `LeaveSessionResponse` PDU is the `leaveSessionResult`. A value of `success` indicates that the leave session service request was processed successfully and the user is no longer a participant of the session (if the session's `sessionNotifiPolicy` is set accordingly, this will be notified to all participants and/or managers of the session). After a successful leave session service the the flows on the GUA side can be removed and the resources reserved for these flows can be released.

A `noSuchSession` result indicates that the `sessionName` specified in the `LeaveSessionRequest` PDU does not exist or that the user is not a participant of the specified session.

3.2.13 Leave Group

The 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 is participant of a session he wants to leave, it is not possible to leave the group without first leaving the associated session.

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

The `LeaveGroupRequest` PDU is sent from a GUA to a GSA. The `bindId` attribute identifies the user who is requesting the leave group service. The `groupName` attribute specifies the group to 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 {
    bindId                      [0]     BindId,
```

```
    leaveGroupResult            [1]     ENUMERATED {
        success                         (1),
        noSuchGroup                     (2),
        noSuchMember                    (3),
        notManager                      (4),
        activeSessionWithinGroup        (5) } }
```

The `LeaveGroupResponse` PDU is sent back from the GSA to the GUA. The `bindId` identifies the user who requested the leave group service. The only other attribute of the `LeaveGroupResponse` PDU is the `leaveGroupResult`. A value of `success` indicates that the leave group service request was processed successfully and the user (or the `leavingMemberName`, if specified in the `LeaveGroupRequest` PDU) is no longer a member of the group (if the group's `groupNotificaPolicy` is set accordingly, this will be notified to all members and/or managers of the group).

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 `leavingMemberName` 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 has to left first.

3.2.14 Delete

The 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.

```
DeleteRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    objectName                  [1]     GmsObjectName }
```

The `DeleteRequest` PDU is sent from a GUA to the GSA. It includes a `bindId` attribute which identifies the user requesting the delete service. The second attribute is the `objectName`, which identifies the object to be deleted.

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

The `DeleteResponse` PDU is sent back from the GSA to the GUA. It contains a `bindId` attribute which is used to identify the user for whom the response is given. The `deleteResult` specifies the result of the delete request. A value of `success` indicates the success of the delete request. In this case, the object specified in the request has been deleted from the GMS. All other values of the `deleteResult` attribute are error messages meaning that the object has not been deleted.

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. One example for this is the deletion of a session which still has participants. Only after all participants have left the session (using the leave session service described in section 3.2.12) it will be possible to delete the session object. Deleting a group is only possible if the group has no more members and no sessions associated to it. On the other hand, if a user is deleted, this will be possible even if he has existing relations (the relations of a user are described in section 2.3.1). He will be removed from these existing relations automatically. Two exceptions are if he is the last owner or manager of a GMS object. In this case, it will not be possible to delete the user.

3.2.15 Renegotiate

The 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. GAP 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 GAP, it is only possible to propagate a set of new QoS values to all flow participants.

```
RenegotiateRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    flow                        [1]     GmsObjectName,
    renegotiatedQosParameters   [2]     SET OF QosRenegotiation }
```

The `RenegotiateRequest` PDU is sent from the GUA to the GSA to request the propagation of new QoS values for a flow. The `bindId` identifies the user who is requesting the QoS renegotiation. The `flow` attribute identifies the flow which is subject of the QoS renegotiation. The `renegotiatedQosParameters` 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 {
    bindId                      [0]     BindId,
    renegotiateResult           [1]     ENUMERATED {
        success                         (1),
        noSuchFlow                      (2),
        notAuthorized                   (3),
        notRenegotiable                 (4),
        noSuchQosParameter              (5),
        nameTypeMismatch                (6),
        illegalRenegotiationValues      (7) } }
```

The `RenegotiateResponse` PDU is sent back from the GSA to the GUA. The first attribute, the `bindId`, identifies the user who requested the renegotiate service. The `renegotiateResult`, the only other attribute of the PDU, informs about the outcome of the renegotiate service. In case of `success`, the renegotiate service request has been accepted and renegotiation PDUs (described in section 3.2.20) will be sent to all flow participants. However, the `success` response only indicates the acceptance of the renegotiate service request, it does not mean that the renegotiation PDUs have already been sent to the flow participants.

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.

3.2.16 Invite

The invite service is used to invite another user to a specific event, which will usually be an application using a session. The user being invited will be notified by a invitation PDU (described in section 3.2.19), which contains the invitation submitted by the inviting user using the invite service. The invitation may either be a GMS invite message or an application specific message, which can contain any data an application needs to process an invitation.

```
InviteRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    invitedUsersName            [1]     GmsObjectName,
    invitedUserGua              [2]     GuaAddress,
    invitationType              [3]     CHOICE {
        gmsInviteMessage            [0]     IA5String,
        appInviteMessage            [1]     SET {
            invitedUsersApplication     [0]     Application,
            message                     [1]     OCTET STRING } } }
```

The InviteRequest sent from the GUA to the GSA contains, as usual, a bindId as first element, which identifies the inviting user. The next element is the invitedUsersName, which specifies the name of the user being invited. The invitedUserGua is necessary to uniquely identify the user being invited, because he may be bound to the GMS more than once. The GuaAddress is used to identify one binding of the user. The invitationType of an invite request may either be a gmsInviteMessage, which is a simple IA5String. If this invitationType is selected, the invitation will be presented by the invited user's GUA. If, on the other hand, the appInviteMessage type is chosen, it is necessary to identify the invitedUsersApplication, which will then get the message as an application specific OCTET STRING. This mechanism is useful for application specific invitation mechanisms.

```
InviteResponse ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    inviteResult                [1]     ENUMERATED {
        success                     (1),
        noSuchUser                  (2),
        noSuchBinding               (3),
        noSuchApplication           (4) } }
```

The InviteResponse PDU, sent back from the GSA to the requesting GUA, first contains a bindId which identifies the user for whom this response is being received. The only other element of the response is the inviteResult, which may indicate a success or failures. In case of success, the GMS accepted the invite request and will forward the invite message to the user being invited. It is important to notive that a success response does not mean that the message was successfully delivered to the invited user.

Possible failures are `noSuchUser`, indicating that the user being invited does not exist (ie the `InviteRequest` specified a non-existing `invitedUsersName`), `noSuchBinding`, indicating that the binding being specified does not exist (ie the `InviteRequest` specified a `invitedUserGua` to which the user is not currently bound), and `noSuchApplication`, indicating that the application specified does not exist at the invited user's binding (ie the invited user did not use the bind application service with this application). In any of these cases, the invite message is not delivered to the invited user.

3.2.17 Manager

The manager service is used by the GMS to query managers (of groups or sessions) about their response to join requests issued by users. Depending on the join request, the user requesting the join is blocked until the outcome of the manager requests is clear (`synchronousTimeout`), or he may proceed and will receive a notification (described in section 3.2.18) when the outcome is clear (`asynchronousTimeout`). All these procedures are only used if the `groupJoinPolicy` (described in section 2.3.2) or `sessionJoinPolicy` (described in section 2.3.5) is set accordingly.

```
ManagerRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    managerRequestId            [1]     ManagerRequestId,
    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 a GSA to a GUA. The `bindId` identifies the user (which has the role of a manager) who should be queried about the join request of a user. Because the `ManagerRequest` PDU may be sent at any time after the user has successfully bound, and because multiple requests may be sent to the user, a `managerRequestId` is included in the request which identifies the request and must be used in the response. The `managerRequestType` attribute contains the actual request data.

If the `managerRequestType` is set to `joinGroupRequest`, the manager has to give a response about a user's request to join a group he is a manager of. The `joinedGroup` attribute identifies for which group the join request has been issued. Because user and groups may join a group (depending on a group's `groupMembers` attribute as described in section 2.3.2), the `memberCandidate` attribute may either specify a `user` or a `group` who want to join the specified group.

If the `managerRequestType` is set to `joinSessionRequest`, the manager has to give a response about a user's request to join a session he is a manager of. The `joinedSession` attribute identifies for which session the join request has been issued. The `participantCandidate` attribute specifies which user wants to join the specified session.

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

The `ManagerResponse` PDU is sent back by the GUA to which the requested manager is bound to the requesting GSA. It contains two identification, the first one being the `bindId` specifying by which user the response is being sent, and the `managerRequestId`, which is used to associate the response with the request sent by the GSA. The `managerResult` may either specify the manager's approval (`approve`) or refusal (`refuse`) of the join request. A third result value is `noMorePendingRequests`, which is signaling to the GSA that the GUA can currently not handle more outstanding manager requests. Typically, this will change after the GUA sent other `ManagerResponse` PDUs (containing `approve` or `refuse` results) to the GSA.

3.2.18 Notification

The notification service is used by the GSA 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 three 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 last class is used to inform users about outstanding results of join services.

```
NotificationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    notificationType            [1]     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 },
        pendingJoinNotification     [8]     SET {
            pendingJoinId               [0]     PendingJoinId,
            joinResult                  [1]     ENUMERATED {
                success                         (1),
                timeout                         (2),
                failure                         (3) } } } }
```

The `NotificationRequest` PDU is sent from the GSA to a GUA to inform a user about an event. The `bindId` is used to identify the user who is to be notified. The `notificationType` specifies which

type of event occured and will be described according to the three 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 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 of the affected session and the session's `sessionNotifiPolicy` attribute is set accordingly.

The last class of notifications is used to inform users about outstanding results of join services. Both the join group service (described in section 3.2.10) and the join session service (described in section 3.2.11) allow users using them to specify the `waitForManagerReplies` attribute with a `asynchronousTimeout` value. In this case, the result of the service is notified with a `NotificationRequest` with a `pendingJoinNotification`. The `pendingJoinId` is used to identify the join service to which the notification belongs (the `pendingJoinId` is returned by the join group and join session service if the `asynchronousTimeout` variant has been specified by the requester). The `joinResult` is either `success`, `timeout`, or `failure`. In case of `timeout` and `failure`, the join request failed.

3.2.19 Invitation

The invitation service is used by the GSA to inform users (bound to the GUA to which the `InvitationRequest` is sent) about invites (described in section 3.2.16) issued by other users. Depending on the invite request, the invitation request can either contain an application specific encoded invitation or a string which is used as an invitation message. The content of the invitation does not imply any actions to be taken by the invited users, such as joining a group or session, although these actions are the main uses of the invitation service.

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

The `InvitationRequest`, which is sent from a GSA to the GUA to which the invited user is bound, first contains a `bindId` field which identifies the user who shall receive the invitation. Because

all users must bind to the GMS using the bind user service (described in section 3.2.2), the GSA is able to identify the GUA and the `bindId` to which the invitation has to be sent. The next field `invitingUser` gives the GMS name of the user who issued the invitation (using the invite service). According to the structure of the `InviteRequest` PDU, the `invitationType` is defined to be a `gmsInviteMessage` or an `appInviteMessage`. The explanations for this attribute can be found in section 3.2.16.

3.2.20 Renegotiation

The renegotiation service is used by the GSA to inform users about a QoS renegotiation. The information being transmitted by the GSA always comes from a user who used the renegotiate service (described in section 3.2.15) to propagate new QoS values for a flow. However, because a actual QoS renegotiation currently is not possible using GAP (it is only possible to propagate new values), the receiving user must accept the new QoS values and modify the respective flow's connection accordingly.

```
RenegotiationRequest ::= --snacc isPdu:"TRUE" -- SEQUENCE {
    bindId                      [0]     BindId,
    sessionName                 [1]     GmsObjectName,
    flowName                    [2]     GmsObjectName,
    renegotiatedQosParameters   [3]     SET OF QosRenegotiation }

END
```

The `RenegotiationRequest` PDU is sent from a GSA to a GUA no notify a user about new QoS values for a flow. The `bindId` is used to identify the user who should be notified about the QoS renegotiation. The `sessionName` and `flowName` attributes are used to specify the session inside which the flow is being changed and the flow itself. The `renegotiatedQosParameters` 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 `renegotiatedQosParameters` attribute of the renegotiate service request (`RenegotiateRequest`) described in section 3.2.15.

References

[1] 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.

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

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

[4] International Telecommunication Union. Association Control Protocol Specification. Recommendation X.227, 1988.

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

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

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

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

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

[10] 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.

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

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