Congress: CONnection-oriented Group-address RESolution Service[1]

T. Anker D. Breitgand D. Dolev Z. Levy

Email: {anker,davb,dolev,zohar}@cs.huji.ac.il
Url: http://www.cs.huji.ac.il/{~anker,~davb,~dolev,~zohar}

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

Technical Report CS96-23

December, 1996

[1]This work was supported by the Ministry of Science of Israel, grant number 854-6195

Abstract

The use of a high bandwidth multicast is becoming widespread in today's network applications. Many of these applications use multicast groups with dynamic membership such as multi-media conferencing, multi-media broadcasting and multi-media distributed data bases. ATM UNI 3.1 and 4.0 protocols offer the point-to-multipoint connection type that enables multicast over native ATM. Point-to-multipoint connections may be utilized for efficient implementation of multicast groups. In both ATM UNI 3.1 and 4.0, however, explicit information about the end-points (members of a multicast group) participating in the multicast connection is required at a connection set up time. Unfortunately, there is no standard mechanism that facilitates the maintenance of such *group membership* information.

In this document we present for the first time a CONnection-oriented Group-address RESolution Service (CONGRESS). CONGRESS incorporates a protocol for efficient maintenance and propagation of group membership information in connection-oriented networks, and thus acts as a complementary service to the ATM multicast mechanism and makes it more usable. CONGRESS does not *replace* the ATM multicast service (*i.e.* does not actually open connections for data transmission), but only resolves group addresses, leaving freedom to the application designer to use the resolved addresses as desired.

Applications that use CONGRESS may name groups arbitrarily, using logical names (group addresses). This enables the application to refer to multicast groups as abstract services. The membership of a group maintained by CONGRESS may change dynamically and is also sensitive to current network connectivity. Groups may consist of a large number of members and may span world-wide. In order to make the service efficient and scalable, CONGRESS services are maintained using servers that are organized hierarchically.

1 Introduction

The concepts of group communication and multicast are basic building blocks of an ever increasing number of communication-oriented applications. The fast networking technology offered by the ATM technology allows for new and more demanding types of distributed and multi-process applications to be implemented. Major examples are multi-media conferencing, pay-TV and high availability replicated data bases with high bandwidth demands such as image data bases. Extensive research is currently done in optimizing scalable reliable multicast protocols to meet the demands of such applications [21, 12, 22]. Many of these applications make use of highly dynamic *multicast groups*. One example is a TV broadcasting service that serves groups of clients that may join or leave constantly. Although ATM network protocols ([4, 5]) enable the formation of multicast connections that may be used by such applications, there is no mechanism for managing membership information of multicast groups.

In this document we present for the first time a CONnection-oriented Group-address RESolution Service (CONGRESS) for efficient maintenance and propagation of group membership information. CONGRESS operates in an ATM environment and manages groups of clients. A CONGRESS client is an application or a transport-layer entity running on an ATM based host. For the rest of this paper we refer to such an application as an *ATM end-point*. An end-point application can use CONGRESS services in order to JOIN groups, LEAVE groups or find out who are the members of a group by issuing a group name RESOLVE REQUEST or through reception of constant updates on group membership. Using this knowledge, end-points may form and maintain point-to-multipoint connections even when the set of receivers in the connection is dynamic. Group addresses (names) are chosen arbitrarily by the applications that form the groups. Thus, addresses are not bound to any fixed addressing scheme.

Multicast groups may consist of a large number of members, which may be geographically far apart. This causes membership to be highly dynamic. A protocol that manages the membership information will be forced to propagate large amounts of membership data across long distances. In order to be scalable, CONGRESS is designed to minimize the network traffic needed to maintain the dynamic group membership. The scalability and efficiency are achieved by using dedicated CONGRESS servers that are organized hierarchically across the global network, and propagate necessary information about multicast groups which have members in their area.

As an address management protocol, CONGRESS does not actually open connections, but only resolves multicast group addresses. Hence, it serves as a complementary service to the ATM multicast connection mechanisms. There is no guarantee, once a membership of a multicast group is obtained, that a physical connection from the resolver through the network to the recipients is possible. This depends on the network congestion level, the Quality of Service (QoS) requested for the connection and possibly the mechanism used to form the connection (point-to-multipoint, multiple point-to-point, etc.).

1.1 Incompleteness of Existing Protocols

The connection-oriented nature of ATM as opposed to earlier wide-spread networking technologies, presents new difficulties in establishing and managing multicast connections. The currently established standards of ATM UNI 3.1 and 4.0 [4, 5] offers the point-to-multipoint connection type to enable multicast over native ATM. Another tool for multicast, a multipoint-to-multipoint distribution tree, is presented in [14] and used for propagating signalling information. There are also multicast protocols that make use of a multicast server to manage multicast connections such as [3] (mainly within subnets) and [12] in which a group communication server is used to collect control information to achieve reliability.

In all of these techniques, an explicit list of the recipients' ATM end-point addresses is necessary for the establishment of the connection. This is true whether the connection is formed directly by an end-point or by an intermediate server. The address list might be obtained in a relatively straightforward manner by applications that work with static predefined multicast groups, using a common data base or known multicast servers. This becomes harder to implement when the members of the group are unknown at the time of connection set-up and even more difficult when dealing with multicast groups which change dynamically. No mechanism for managing dynamic multicast group addresses in ATM is available so far.

When connected to a dynamically changing group of receivers, the end-point that formed the connection may need to receive updates on the membership of the multicast group in order to update the connections as needed. Updates on membership are necessary especially when only root-initiated join is possible, as in the UNI 3.1 [4] signaling protocol. However, even when leaf-initiated join is possible, as is in the UNI 4.0 [5] standard, the joining party must know the VC assigned to that specific connection, which is less abstract then a logical group name. Moreover, in order to become connected to all members of the group, the new joiner must know the VCs of the multicast trees of all group members.

The services provided by CONGRESS are aimed to solve exactly these problems. Whenever a connection to the group members must be formed, a resolve request is issued to CONGRESS by the root of the connection (which may be either a multicast server or an end-point). As a reply to this request CONGRESS returns a list of the members' addresses that can be used for establishing a connection. Whenever a new member joins or leaves the group, the root of the point-to-multipoint connection receives an update and can include/discard the new member in the connection.

It should be noted that UNI 4.0 does provide group addresses, which are used to support the ATM anycast service (see [6]). However, this service allows a user to request a point-to-point connection only to a single ATM end-point that is a member of the group. The user is not provided with a mechanism to resolve the ATM group address into a list of end-point members. Moreover, the anycast service does not force the ATM signalling mechanism to support such a resolve operation.

1.2 Motivation for CONGRESS

In this section we give examples of the application that will benefit greatly from the use of a low-level native ATM protocol like CONGRESS for maintaining dynamic multicast group membership.

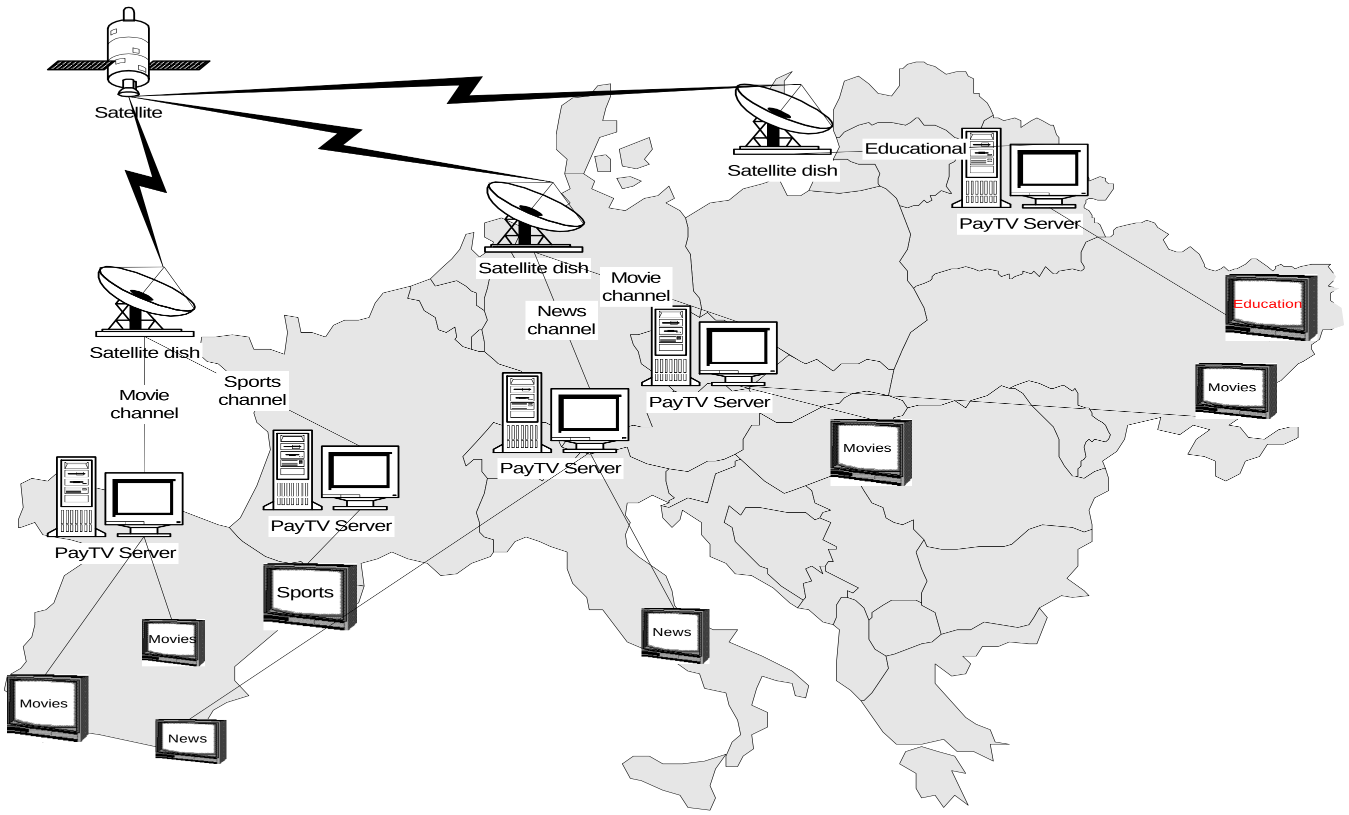


Figure 1: An example of Pay TV Broadcast services in Europe mainland.

Pay TV Application

Consider a world-wide television broadcasting service in which a group of servers receives broadcast signals from a geo-stationary satellite. The servers propagate it to the clients at home through the ATM network with high QoS using point-to-multipoint connections. Many different services can be provided at any time, e.g. several simultaneous programs. For the purpose of load-balancing, each service is supplied only by some of the servers. In addition, each client can receive the service only from those servers to which there is connectivity through the network. And finally, to save bandwidth, only clients that have their televisions open and tuned to a specific service should receive signals for that service. Figure 1 depicts a layout of the service. This scenario raises three issues:

- the clients need to notify some server providing the service about its desire to receive the service;
- the servers need to maintain high bandwidth point-to-multipoint connections to dynamically changing groups of clients;
- the servers need a mechanism for load balancing.

The first two problems may be resolved in framework of an ATM anycast service provided by UNI 4.0 [6]. However, ATM anycast service provides its user with point-to-point connections only. In order to achieve efficiency, native ATM multicast connections should be used. Additional steps that are beyond the ATM anycast service should be taken in order to add a client as a leaf in the server-rooted multicast tree. Another limitation is the impossibility to provide a load-balancing algorithm among the servers. Usage of the ATM anycast will override such an algorithm, and the network will choose a server based on its own considerations. These limitations make the ATM anycast service inconvenient for development of the Pay TV application.

CONGRESS provides a dynamic multicast group abstraction that facilitates the solution of the three problems stated above. In the above example, the broadcasting service could be represented by a logical group name called "PayTV", for example, which a client can join to receive the service without knowing which and where the active servers are. An active server would resolve the clients' addresses through the group name "PayTV" in order to establish a point-to-multipoint connection to the clients.

Video Chat Application

As a second example, consider a video chat application, like CU-SeeMe [15], implemented over native ATM. The most natural mechanism for the data transfer in such application would be multicast. The problem, however, is that the membership of a group of participants is not known in advance. It is impossible to open a point-to-multipoint connection in ATM, if the ATM addresses of the participants are not known. Note also that this information is dynamic, because participants may join and leave the chat sessions arbitrarily (like in the IRC application or the Netscape Chat).

Working with CONGRESS an end-point that wants to join an arbitrary video-chat group can request upon joining, the ATM addresses of the other end-points participating in the chat. Using the received address list, the new member can form connections to the other members with any desired QoS and start gossiping. At the same time, the other members of the group will receive a notification about the new member and add him to their own connections so he can hear their own gossip...

IP multicast over ATM

Current internetworking is based primarily on IP [17]. This protocol will continue to play an extremely important role in the future due to an immense amount of applications that were designed to use it. An efficient implementation of IP over ATM is considered crucial for final success of ATM as a universal communication infrastructure. Extensions to IP [18, 16] provide a mechanism for multicast data transfer in IP networks. The implementation of IP multicast over ATM WAN emulates the connectionless IP routing protocol over connection-oriented ATM network. The underlying ATM routing mechanisms are not utilized by IP. Thus, IP multicast routing is suboptimal in comparison with the native ATM routing.

The implementation of IP multicast over ATM may be substantially improved using protocol like CONGRESS. This can be done using CONGRESS at the multicast routers that connect subnets or LANs to an ATM backbone. Upon the reception of an IP multicast packet destined to a multicast group, the router can use CONGRESS in order to obtain the list of the ATM addresses of the multicast group members[1]. Then it can use native ATM multicast mechanisms in order to efficiently multicast this packet. Using CONGRESS, the incorporation of QoS support into IP multicast over ATM will be relatively straightforward, given that the non-ATM components of the internet support them.

1.3 Sensitivity to Failures and Network Partitions

A notification upon a member joining or leaving a group, may be useful at the application level. Membership maintenance is needed for purposes such as bookkeeping or flow-control adjustments. Such notifications are not always possible to implement at the application level. For example, if a member of the group fails or becomes disconnected due to a network partition, it can not inform the rest of the group about this event.

Handling such cases requires the resolution protocol to be sensitive to the availability and connectivity of the group members. This means that from an end-point's point of view, a reply for an address resolution request for a group address should not include addresses of crashed or disconnected group members. Counting disconnected or crashed end-points as active members of the group may have undesirable effects. For example, if a "PayTV" server fails to connect to an active member, it must periodically retry to form the connection in order to provide good services. If, on the other hand, the server "gives up" on disconnected registered clients, it is the clients who would need to constantly try to re-join the multicast group in order to receive the service. These scenarios may burden both the application and the network (with signaling requests). Maintaining multicast group membership that is sensitive to network configuration changes takes this load off the applications and leaves it to the group address resolution protocol.

A network partition can be caused by switch or link failures. Such a partition divides the network into several *components*. In an environment where partitions and failures occur, a more elaborate definition for the multicast group is needed. A *multicast group* is a group of end-points that have all registered into the group by *joining* it. An end-point that joined a group becomes a registered *member* of the group. An end-point may cease being a member of a group by either explicitly *leaving* it, or due to a crash. The *membership* (or *view*) of the group is an addresses list of the members of the group . The membership may be seen differently by each member at different times due to dynamic changes and network partitions. For example, if a, b and c are members of a group G but b belongs to a different network component than the component to which a and c belong, then the membership of G as seen by a and c will be $\{a, c\}$ while b's view of G's membership will be $\{b\}$.

A well known result presented in [20] proves the impossibility of reaching agreement in an asynchronous system. However, many distributed applications require agreement on group membership among group members in order to operate correctly. Practical systems circumvent this problem by taking a presumably faulty entity out of a view [1, 19]. However, in a WAN environment, any kind of distributed agreement is prohibitively expensive in terms of latency. CONGRESS membership service does not guarantee agreement on membership between group members. CONGRESS provides relaxed guarantees on membership that will be described in Section 4.

[1]The members can be either hosts that are ATM based, or routers that connect ATM network with other types of networks.

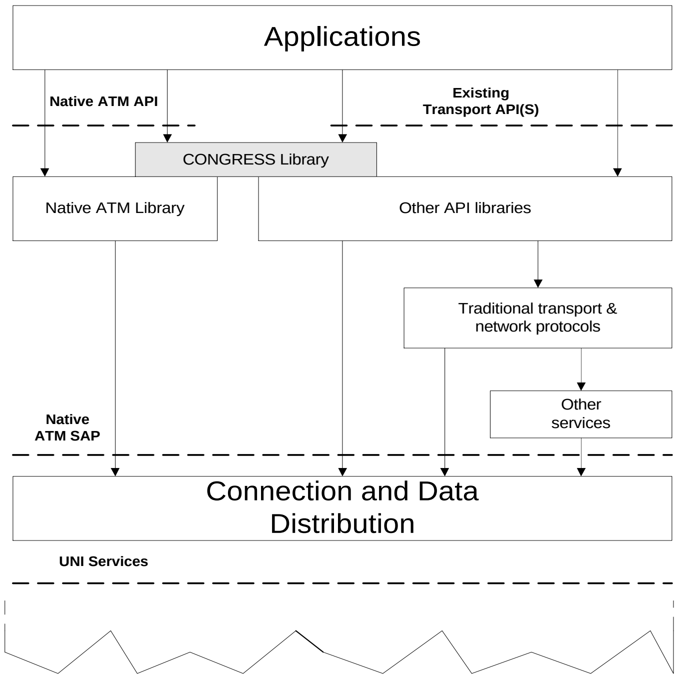


Figure 2: Position of CONGRESS library in the reference model model for native ATM^2.

2 CONGRESS Overview

CONGRESS is an address resolution protocol for native ATM environment. It may be viewed as a sub-layer between the network layer and the transport layer services in the ISO OSI seven-layer model [17]. The location of the CONGRESS library in the ATM reference model is depicted in Figure 2. A CONGRESS client, which may be either a transport-layer entity or a user application running on an ATM based host, can resolve logical group names into a list of the ATM addresses of the group members.

An end-point may become a group member by joining the group or cease its membership by leaving the group. An end-point that is a member of the group may choose to receive on-line updates about the group membership changes.

CONGRESS services are supported by local and global membership servers. A Local Membership Server (LMS) resides in each host and serves as a front-end for the end-points using CONGRESS on that host. Global Membership Servers (GMS) are organized in a hierarchical structure throughout the network, and may be located on dedicated machines or switches. GMSs provide processing of aggregate information and efficient dissemination of CONGRESS messages to the LMSs.

Since CONGRESS operates in an asynchronous environment, it can never supply accurate information about the group membership. Some membership information may be delayed in the network and by the time an end-point receives a resolution reply for a group membership, the real membership of the group may have already changed. This is true for any protocol working in an asynchronous environment [20]. CONGRESS guarantees that if the group membership stops changing at some point, then eventually any resolution request for the

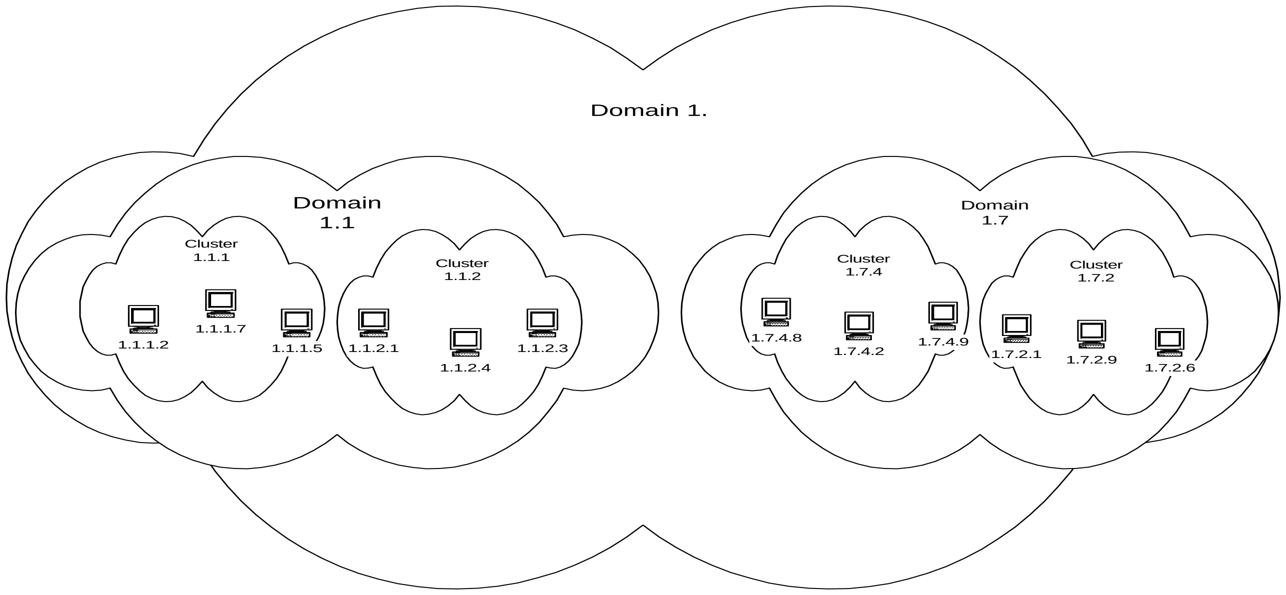


Figure 3: Addresses scoping

group name will result in the *correct* list of addresses for the members of this group. This is explained in detail in Section 4.5.

3 Environment

We assume the existence of a global hierarchical addressing scheme that enables to relate nodes (switches and ATM based hosts) to domains and sub-domains by their ATM address prefixes. The addressing scheme proposed by the PNNI signaling protocol [8] that divides the hosts and switches in the network into *peer groups* based on ATM address prefixes is an example of such a scheme. However, since we cannot be sure that this scheme will indeed be used wholly or partially for internet addressing, and since CONGRESS does not depend on internal features of PNNI to operate, we state here the abstract features CONGRESS would require from any addressing scheme.

We assume that, similarly to the way that today's IP addressing scheme divides the Internet, the network is divided into a hierarchy of *domains*. At the lowest level, a domain consists of a group of hosts and switches. A group of several neighboring domains comprises a higher level domain and so on. The highest level domain could be seen as the whole net. In order to distinguish between the lowest-level domains (which are comprised of hosts and switches) and the higher-level domains (whose members are lower-level domains), we shall refer to the lowest-level (level 0) domains as *clusters*. Figure 3 presents an example of a domain hierarchy. We also assume the existence of a global signaling protocol such as the PNNI signaling protocol that enables connection set-up across the network.

Reliable point-to-point communication is assumed between the CONGRESS servers. This

[2]The full reference model for native ATM services (Native ATM SAP) is given in [7].

can be achieved by using any reliable unicast protocol between the neighboring entities[3].

The network itself (links and switches) as well as the hosts are not assumed to be failure-free. Network partitions/re-merges, and host crashes/recoveries may occur, dividing the network into several components. A network partition can be caused by switch or link failures. Hosts residing in the same network component are physically connected (through links and switches) and may form connections as the signaling protocol allows. Hosts belonging to different components are physically disconnected and no data or signaling information can be exchanged between them. If a host on which a CONGRESS server is running crashes, the services given by that server are no longer supplied. Failed hosts or CONGRESS servers may recover at any time.

In order to make the protocol sensitive to such events, each CONGRESS server receives reports from a *fault detector* module that monitors the network. The fault detector module provides reliable notifications in case of host/process crashes and recoveries. All CONGRESS communication passes through this module. In any implementation of this module, we require that that protocol messages received from an entity (host/process), will be considered as an indication for this entity's liveliness. Thus, if an entity is considered faulty, and a message is received from this entity, the fault detector module delivers a notification indicating liveliness of the entity **prior** to the delivery of the entity's message.

There are various ways in which fault detector modules can be implemented. Common ways are *polling* and *heartbeats*. In [25] a framework for implementing a fault detector in wide area networks is described. In any case, it should be noted that in an asynchronous environment where failure detection is based on message exchange or timeouts, if a machine monitoring a peer machine for liveliness suspects a failure, it can never be sure whether it has been partitioned from the peer (link failure), if the peer has failed or if the peer is just too slow. A slow machine may be suspected by the fault detector module as failed and on the other hand, delayed messages in the network may cause the fault detector to report the recovery of a machine that is actually failed or disconnected. In either cases, we assume that if the state of the host/process whose liveliness has been misinterpreted remains constant, the fault detector will eventually correct itself and send a report which reflects the real state of the host/process.

4 CONGRESS Services and Guarantees

4.1 Logical Group Names

In order to allow easy dynamic group formation for specific purposes, groups are named by *logical names* that are specified by end-points. This is different from other existing schemes where a fixed group address space is provided. A logical-name addressing scheme allows more flexibility in giving meaningful names to groups and sets no limit on the number of possible groups.

Groups supported by CONGRESS can be either *permanent* or *spontaneous*. Permanent

[3]Limitations of TCP as a reliable protocol over WAN are now recognized and a number of smart protocols implementing reliability between neighboring communicating entities at low cost have appeared recently [23, 13]. Reliability could also be implemented more efficiently in CONGRESS itself to suit its requirements but such an implementation will not be described in this paper.

groups have a well known registered logical name, and they exist even if they have no members. They can be formed only through administrative agreements. Such groups can be used to support well-known services (e.g newsgroups). In contrast, spontaneous groups are intended to be used for private or temporary purposes. An end-point *forms* a spontaneous group by being the first end-point to JOIN it. The name of the group is specified by this end-point with a JOIN request. Any end-point issuing a JOIN request with this group name afterwards, does not form a new group but joins the existing group. When a spontaneous group contains no more end-points, *i.e.* all end-points that have joined the group left it, it ceases to exist. Both types of groups, permanent or spontaneous, can be dynamic in terms of membership, meaning that end-points can join or leave groups at any time.

The distinction between permanent and spontaneous groups is made in order to protect groups used for well-known services from accidental use of their names for private purposes. The use of a group for more than one service can harm application protocols that assume that only relevant messages flow in the group. Malicious end-points could still join permanent groups and attack the protocol executed in them through redundant messages. Additional security measures must be taken against such possibilities, but this is out of the scope of this paper.

In order to support permanent groups, databases for permanent group names must be managed and kept consistent at the different CONGRESS servers. The way this is done will not be described in this paper. A general approach for maintaining distributed databases can be found in [26]. In the rest of this paper, all groups will be related to as spontaneous unless stated otherwise.

4.2 Group Scoping

The distinction between permanent and spontaneous groups does not solve the problem of possible overlapping of spontaneous groups. End-points may desire to form unique spontaneous groups for their own specific use. If two end-points accidentally specify the same group name, they may unintentionally find themselves in the same group. The probability that such scenarios will occur grows as the network becomes larger and more end-points participate in the protocol. Although a name-locking mechanism could be implemented (e.g., using a two-phase commit protocol [9]), this mechanism incurs a high overhead at connection set-up time and should definitely not be forced on all end-points that use CONGRESS.

In order to reduce the probability of such scenarios, an end-point that forms or joins a group must specify a *scope* in which the group name will be advertised. The scope is a network domain defined in the network's domain hierarchy as specified by the network's addressing scheme [4]. The pair (G, S) where G is a group name and S is a scope, is unique: CONGRESS refers to a group G formed in scope S_1 as a totally different group from another group G formed in scope S_2 if $S_1 \neq S_2$, no matter what the relations between S_1 and S_2 are (subsets of one another, intersecting or disjoint). Thus, if two end-points make a call to form group G in the same scope S, they will become members of the same group while if one end-point forms G of scope S_1 and another end-point forms group G of scope S_2 they will form two different groups. An end-point must reside in the scope of any group that it

[4]If the underlying network hierarchy is the PNNI hierarchy, then the CONGRESS scoping may match the PNNI peer group scoping mechanism

forms or joins.

Similarly to the spontaneous groups, permanent groups are also declared in a certain scope. This allows well-known services to co-exist with other (permanent or spontaneous) groups bearing identical names, but declared with a different scope. In the rest of this paper we shall denote groups by their logical names only without specifying their scope unless it is relevant.

In addition to name uniqueness, using scopes reduces network load and CONGRESS service latency. The group scoping mechanism restricts the CONGRESS traffic (*i.e.* the group membership data propagated among the involved end-points) to the relevant parts of the network. This prevents a local group from flooding the rest of the network. In the television broadcast example, a station that broadcasts only in Jerusalem could use a multicast group called "PayTV-Jerusalem" whose scope is Jerusalem only. The bandwidth reduction achieved by scoping can be significant since data communication often follows locality patterns. Traffic *within* a properly organized communication domain is more intensive than the traffic that streams in and out of the domain. This implies that the number of local groups, whose members are located relatively close to one another might considerably exceed the number of global groups spread over wide distances.

4.3 Membership Notifications

Group membership may be dynamic, *i.e.* members may join or leave groups arbitrarily. In addition, group membership is affected by network partitions/remerges, and hosts crashes/reincarnations as explained in Section 3. A member *memb* of a group G may derive G's group membership locally using messages received from CONGRESS. We shall refer to these as *membership notifications.*

CONGRESS provides two types of membership notifications: *absolute* and *incremental.* An absolute membership notification for a group G contains a complete list of the ATM addresses of the end-points that are members of G. This type of notification is provided by CONGRESS in response to user's request (e.g., RESOLVE request). An incremental membership notification, or *membership update* for a group G reports of a **change** that occured in the membership of G and holds only minimal information needed to depict the change. This type of notification is received automatically from CONGRESS each time the membership of G changes. A user may optionally request incremental notifications upon joining G. Note, that the first incremental notification received by an end-point upon joining the group G, will be exactly the same as the absolute one.

An incremental notification holds a list of ATM addresses. The addresses in this list can come in two forms:

- Host/Domain address: An ATM address prefix describing a domain in the domain hierarchy. All addresses of the hosts and switches in this domain share this common prefix. The domain can consist of a single host, in which case the address would be a full ATM address of a host;

- End-point address: Consists of a host address and an identifier of the end-point process on that host (e.g., port number, VCI).

Incremental notifications are subdivided into special subtypes for reporting joining and leaving of the individual end-points, and domain failures. In order to keep the size of the incremental notifications minimal, the address list is kept short by specifying domain or host addresses instead of the individual end-points addresses wherever possible. The following cases are possible:

1. When end-points join a group due to either JOIN request or a network merge, the update contains the list of the new end-points. Note, that when a group remerges (due to a network merge), end-points that came from different components will receive complementary incremental notifications;

2. When members leave a group (using the LEAVE request) or crash, the update message contains end-point addresses of these members;

3. If a CONGRESS server crashes or the network partitions so that certain domain [5] becomes disconnected from the rest of the CONGRESS servers, we say that there is a partition in terms of the CONGRESS services. This means that the CONGRESS servers in different network components can not interact. The update message generated following such an event, holds a domain address of the disconnected domain.

   If the end-point that receives an incremental update belongs to the disconnected domain itself, it should discard all the end-points that are located outside of this domain. Otherwise, it should discard all the end-points that belong to the failed domain. These two cases are treated by two different types of the incremental notifications, called FILTER_IN and FILTER_OUT respectively.

In the first two cases above, the name of the group in which the change occured is specified on the membership notification. In the third case, no group name is specified. To hold the necessary information, each membership notification consists of three fields:

1. **Notification Type**: This field can hold one of the following values:

   - **absolute**: The notification is absolute, *i.e.* a reply to a resolve request;
   - **ep_join**: The notification reports of end-point(s) joining a group. In case of domain re-merging, end-point will receive a notification of this type holding a list of all the members of the group in that domain;
   - **ep_leave**: The notification reports of end-point leaving a group or failing;
   - **filter_in**: The notification reports of a domain partition. The receiving end-point belongs to the reported domain, and this domain is disconnected from the rest of network in terms of CONGRESS services;
   - **filter_out**: The notification reports of a domain partition. The receiving end-point does not belong to the reported domain, and all end-points in this domain are disconnected of the receiving end-point.

2. **Group**: The group whose membership is reported, if the notification type is **absolute**, **ep_join** or **ep_leave**;

3. **Address List**: A list of one or more end-point addresses, or a domain address.

[5] A domain may consist of a single host on which end-points are running.

4.4 User Interface

A client of the CONGRESS services may be a transport layer that supports multicast or a user application that needs to determine the membership of a group. This group may consist of servers that provide a service, clients that receive a service, or a set of entities cooperating in a distributed operation. The functions that are provided by CONGRESS are:

1. RESOLVE(G, $[S]$): Resolution of a logical group name G of scope S, if S is specified, into a set of the ATM communication end-points. The caller receives a *resolve-reply* that is an absolute membership notification for G as described above. If S is not specified, the reply resolves the addresses of all groups named G in all scopes that contain the caller;

2. JOIN(G, S, $Online\_flag$, $Group\_type$):

   Become a registered member of the group G in scope S. The calling end-point must reside in the scope specified by S. An end-point may join more than one group.

   $Online\_flag$ can be either SET_ONLINE, or RESET_ONLINE. Setting/resetting this flag enables/disables the reception of incremental membership notifications (*membership-updates*) in the joined group. In addition, if this flag is set to SET_ONLINE, then it is guaranteed that the end-point will automatically receive one absolute membership notification. Note that incremental membership notification might be received before reception of this absolute membership notification.

   $Group\_type$ can be either PERMANENT, or SPONTANEOUS. If $Group\_type$ is set to SPONTANEOUS, then the first process that joins the group forms it. If $Group\_type$ is set to PERMANENT, but group G of scope S is not registered as a PERMANENT group, then the JOIN call will fail and return an appropriate error to the user. If $Group\_type$ is set to SPONTANEOUS, and group G of scope S is already known to CONGRESS as a PERMANENT group, then the JOIN call will fail with an appropriate diagnostics;

3. SET_FLAG(G, S, $Online\_flag$): Enable/disable the reception of incremental membership notifications (*membership-updates*) in the group G of scope S while being a registered member of G. The values of the $Online\_flag$ are the same as in JOIN above;

4. LEAVE(G, S): Terminate a membership in group G of scope S. If G is a spontaneous group it will cease to exist when the last member leaves it. If G is a permanent group it will continue to be known to CONGRESS even if it has no registered members left as a result of the LEAVE call.

It is up to the end-points that have their SET_ONLINE flag set to maintain the exact group membership derived from membership-updates. An end-point can maintain the group membership by obtaining an absolute membership notification for the group and then update this membership view according to the incremental notifications that it receives. Each end-point can be associated with a single host or a single domain according to its address. So, if for example, an update about a domain failure is received, the end-point can delete all group members that belong to that domain from its membership view of the group.

Membership in a group (achieved by the JOIN function call) does not necessarily imply the existence of an active ATM connection to the other members of the group. An end-point which is a member of a group may or may not open any kind of connection to other members of the group. It may open UNI 3.1/4.0 point-to-multipoint connections or separate point-to-point connections to other members of the group. This enables adaptability to any future multicast connection standards defined. In addition, no restriction of QoS of connections is implied by CONGRESS.

4.5 CONGRESS Guarantees

Membership notifications can be retrieved either by absolute notifications (which sometimes will be referred as *resolve-replies*) or by incremental notifications as described in the previous section. In this section we describe the guarantees CONGRESS supplies on membership notifications.

Membership updates and replies to resolve requests should be received in an order that reflects the order in which the membership events (or membership changes) occur, in order to enable an end-point to construct an up-to-date view of the group membership based either on membership updates or resolve replies. The ordering of membership events is defined only on events which involve a particular host, domain or end-point. We guarantee this behavior through *per-source chronological ordering of membership events.*

Since the network is asynchronous and protocol messages may be delayed, membership information at different CONGRESS servers at any given time may differ. It is therefore difficult to make many strong guarantees about the correctness membership notifications during a time of instability. If, however, the membership stops changing, then eventually all the members will have the *correct* view of the membership as defined below.

4.5.1 Per-Source Chronological Ordering of Membership Events

Every membership event can be related to a specific *source.* A source can represent either an end-point or a host/domain. A membership event involving a source s in relation to a group G can be either:

1. A JOIN request for group G issued by an end-point s.
2. A LEAVE request for group G issued by an end-point s.
3. A failure of a host/domain s.
4. A recovery of a host/domain s.

A member end-point *ep* of a group G may not receive *all* the membership notifications concerning G. If the network is partitioned, *ep* can acquire notifications only about membership change events in G that occur in the network component to which it belongs. In addition, if *ep* does not have the SET_ONLINE_FLAG set, it will receive membership notifications only through the RESOLVE operation. The guarantee made here relates only to those membership notifications that are actually received by *ep*.

Per-source chronological ordering of membership events guarantees that any two membership events involving the same source will be reported to any end-point in the same order they were occured. Thus, membership notifications on any event are guaranteed to be delivered to other group members in this order. In addition, an end-point that issues several RESOLVE requests, will receive the resolve-replies in an order that reflects the partial order[6] of membership change events. In particular:

- Join/Leave events with respect to a group G initiated by an end-point, will be reflected to all the receiving end-points in the same order that they were issued (with **ep_join/ep_leave** incremental notifications). For example, if an end-point m_1 joins a group G and later leaves it, any other member m_2 receiving both notifications on these events will receive them in the order they were generated. If this was not the case, m_2 would have seen m_1 as a member of G until it issued another RESOLVE request.
- Failure/recovery events of an end-point or a host/domain as detected by the fault detector module will be reflected to all the receiving end-point in the same order of detection (with **ep_leave/filter_in/filter_out** messages).
- Two consecutive resolve to a group address G will be answered with two resolve replies r_1 and r_2 respectively, such that r_2 is delivered to the requesting end-point after r_1. If $r_1 \neq r_2$, then some membership changes occurred between the serving of these two resolve requests by CONGRESS. In such case r_2 is considered more updated.

4.5.2 Correctness of Membership

Here we present a definition of the concept of *correct* membership. However, a formal proof of the correctness of the membership supplied by CONGRESS is out of the scope of this article.

As mentioned above, it is difficult to define correctness of membership in a dynamic asynchronous environment because of network propagation delay. Although messages may be delayed for an unbounded time interval in the network, we assume that this delay is always finite. Any message sent will be eventually received by its destination unless the destination fails (this relies on the assumption of link reliability stated in Section 3).

Due to the propagation delay, an end-point may receive a membership notification msg that reflects obsolete information. If other membership notifications were originated after msg's origination, msg would hold out of date or non accurate information. The newer, more updated information would be received only later, in consequent membership notifications. In particular, at a certain point in time, different members of the same group G in the same network component could have different membership views of G because they have not received the latest membership notification (although all notifications would be received eventually by all members in the component).

A correct membership could only be defined in a state of *stability*. Stability associated with a time T is defined to be the state of the network starting at T (called *stability time*) after which no new incremental membership notifications are originated and all the previously sent notifications have been propagated throughout the network. The stability

[6]Membership change events can be ordered in a partial order according to chronological order in which they occured with respect to a particular source.

associated with T is violated when a new membership notification is originated after time T.

At a stability state associated with a stability time T, a different view is correct for end-points in different network components. A *correct* view of the membership of a group G in a network component C as seen by a member *memb* consists of all the end-points that belong to C that have issued a JOIN(G,...) request before T and have not issued a consequent LEAVE(G,...) request before T. Naturally, if all members of the group G in component C hold the correct view of G after T, they all hold the **same** view of G as well.

In order to ensure that CONGRESS supplies non-trivial membership notifications, we guarantee that the membership computed by an end-point using the membership notifications will be correct once the network reaches a state of *stability*. Any RESOLVE(G,...) request issued by a member *memb* of G in C will receive a *resolve-reply* that holds the *correct* membership of G in component C.

5 CONGRESS Architecture

As mentioned earlier, CONGRESS servers are placed in the network in a hierarchical structure. This structure serves two purposes:

- Support scoping of groups.
- Make the protocol scalable up to serving a world-wide network.

In order to support scoping, the hierarchy that is used must be meaningful to the user. A natural choice would be to use the addressing hierarchy that would be used in future ATM internet. CONGRESS servers are placed in the network according to the hierarchy described in Section 3. A Local Membership Server (LMS) is placed in each host, and serves as an interface to the CONGRESS's services for end-points running on its host. The LMS receives end-point requests for registration to groups, leaving groups and resolving group addresses. The LMS processes them by interacting with the GMS and provides the end-point with replies. The GMSs are placed in a hierarchy based on the addressing hierarchy. A single GMS is placed in each cluster to exchange protocol messages with all the LMSs residing in the cluster. At level i in the domain hierarchy, each domain consists of level $(i-1)$ sub-domains (which are clusters for $i = 2$). Each level $(i-1)$ sub-domain is represented by a GMS. One of these GMSs is chosen to represent the level i domain. This GMS is called the GMS *leader* (GMSL) of this domain[7]. The GMSL of a domain is responsible to communicate and process any protocol traffic flowing in and out of the domain. A single host may execute GMSs of several consecutive levels. This means that if a GMS who is the GMSL of a domain D of level i runs on a specific host, then the GMSL of the level $i+1$ (that contains D) may be executed on the same host[8].

We refer to the GMSLs of level $i-1$ that belong to the same level i domain as a *siblings set*. The level i GMSL selected from this set is termed *parent* of the other GMSLs. The level

[7]This is similar to the way *peer group leaders* are chosen in PNNI.

[8]There is more than one possibility for implementing multiple GMSs on a single host. One way of doing this is to use a single process that acts as multiple GMSs - one for each (consecutive) level managed by the host. Another simpler way, is to use multiple processes, each for a different GMS on a different level.

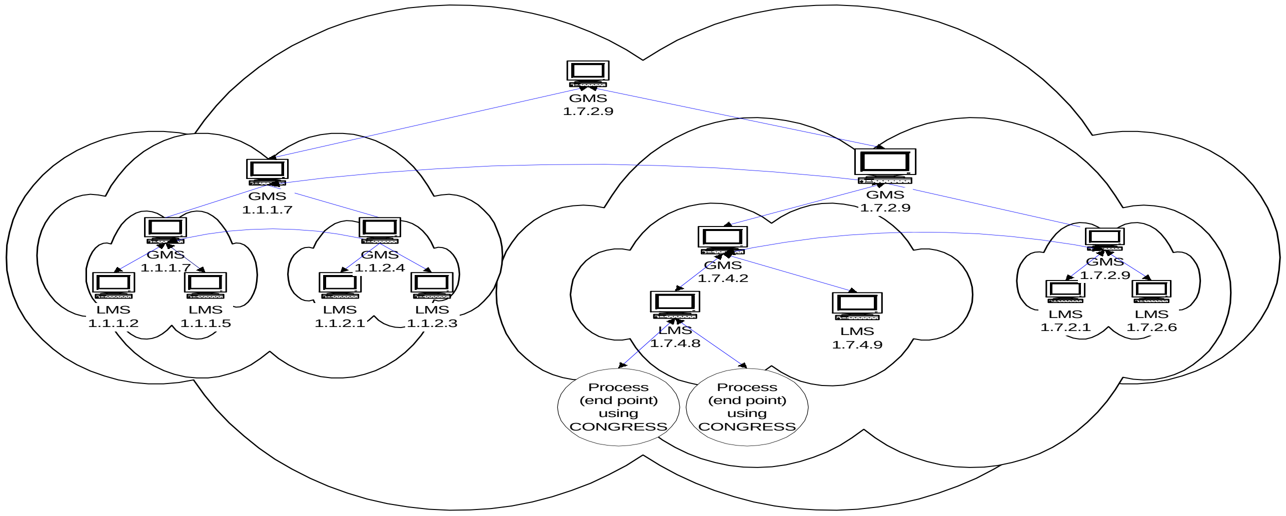


Figure 4: The CONGRESS Protocol Hierarchy Structure

$i - 1$ GMSLs are *children* of the level i GMSL. At the lowest level of the hierarchy LMSs are referred to as children of their corresponding GMS. An end-point belongs to a *domain* of a GMS, if the LMS running on its host is an descendant of that GMS. The domain of an LMS includes all the end-points running on its host. A sample CONGRESS' hierarchy is depicted in Figure 4. The addressing hierarchy in this figure is the same as in Figure 3. Note that GMSL of a domain D at level i is also a GMSL of all the domains that descend from D. This is expressed in the figure by associating the same address with a GMSL in all the levels at which it serves as a GMSL.

5.1 Local Membership Server (LMS)

The LMS serves as a host's service access point to CONGRESS. An end-point that is interested in a CONGRESS service (issuing a join or leave request for a group or resolving an address of a group) submits its requests to the LMS. All replies and notifications about configuration changes (e.g. other host failures) are forwarded to the end-points by the LMS.

An LMS runs at each host participating in CONGRESS. It maintains a membership table that contains a list of the local end-points for each multicast group about which it has knowledge. The LMS knows only about groups that have at least one member running at the LMS's host. Each LMS is connected via a point-to-point reliable connection to its parent GMS. Each GMS has a point-to-multipoint reliable connection to all its children, the LMSs of its cluster.

5.1.1 LMS's Basic Operation

The LMS reacts upon receiving a request from an end-point running on its machine, a message from the GMS running in its cluster, or feedback from it's fault detector module.

When an end-point issues a JOIN, LEAVE, RESOLVE or SET_FLAG request the LMS notifies the GMS of its cluster of the request. RESOLVE request are recorded by the LMS so that when a resolve-reply is received for them, the LMS will know to which end-point to forward the reply. When a failure of an end-point is detected by the LMS it notifies the GMS as if this end-point has issued a LEAVE request.

The messages an LMS receives from its GMS can be either RESOLVE requests originating in a distant machine or reports in the form of the membership notifications described in Section 4.3. When receiving a RESOLVE request for a group G from the GMS, the LMS replies with a message holding the list of end-points belonging to G that it serves. When receiving a membership notification, the LMS informs interested local end-points. If the notification reports a membership update resulting from the failure of one or more hosts or upper-level domains, each process receiving this update may loop through all the groups that it is a member of, and remove all the endpoints related to failed hosts or domains from the group membership. This way, all the calculations of group membership are done locally, at no additional network traffic.

When a host running an LMS recovers from failure, or a new host starts using CONGRESS services, it naturally has no active end-points running, so the recovered LMS does not inform its parent GMS about its recovery. The GMS will detect the recovery through its fault detector. However, when the host has merely been disconnected and now re-merges, the LMS must inform its parent GMS of all the end-points running on the host and a notification is propagated to all the appropriate LMSs.

5.1.2 LMS's Data Structures

The most important data structure of the LMS is a table of the multicast group names where each entry points to a list of the end-points belonging to the domain of the LMS. As described above, the LMS's domain contains only the end-points local to the host of the LMS. Therefore, the LMS's table can be kept reasonably small.

A second data structure held by the LMS is a queue of open resolve requests. This queue holds data on any resolve request sent by the LMS that was not answered yet. It holds the *id* of the end-point that initiated the resolve request and the message *id* with which the LMS stamped the resolve request. The message *id* is a combination of the LMSs address and a sequence number. This *id* will be used by all the servers in order to distinguish between different resolve request, and the replies to them.

Each LMS maintains also a variable MY_GMS that holds the ATM address of the GMS of the cluster to which this LMS belongs. At the initialization time the LMS reads the value of this variable from a configurational file.

5.1.3 A Note On Implementation

Although we stated earlier that each host receiving CONGRESS services will have an LMS running on it, exceptions can be made. For example a host that does not supply a multi-tasking environment can connect to another host (preferably one that is in the same cluster) that is executing an LMS and receive CONGRESS services from it. This solution is also suitable for a group of hosts that typically run few processes using CONGRESS at any time, and it is wasteful to overload each host with an LMS.

Note however that if an LMS will run on each host, it will be easy to exploit the CONGRESS architecture for the implementation of a fault detector within CONGRESS. The LMSs could use OS services to verify the liveliness of end-points running on their machines in a deterministic manner. This information could be propagated through CONGRESS servers to any remote end-point wishing to monitor the liveliness of end-points on that host. This is similar to the layout described in [25].

5.2 Global Membership Server (GMS)

The GMS is responsible for the processing of group address resolution requests, recognition of configuration changes and propagation of protocol messages. In order to prevent flooding of the network with protocol messages, the GMS makes intelligent use of data structures in order to forward messages in the network in relevant directions only.

At the lowest level of the GMS's hierarchy each GMS maintains a list of the active LMSs. When an LMSs fails or recovers the GMS will recognize the event through its fault detector module and inform through the GMS hierarchy all the interested LMSs about the new membership. Note that CONGRESS treats a failure of an LMS as if the LMS's host failed. This is done since the end-points on that host can no longer receive CONGRESS services.

An important characteristic of the GMS is that the data it maintains is *stateless*. It does not hold any data about group membership but only propagates messages. Only information as to *where* to propagate messages is kept by the GMS. The benefit of such a design is that it enables quick and simple recovery of failed GMSs. A recovered GMS would simply forward data in *all* directions until it receives a clearer picture of group membership. This feature is described in further detail in the next section.

5.2.1 GMS's Basic Operation

.

Similarly to the LMS, the GMS's actions are triggered either by messages received from neighboring servers [9] (LMSs or GMSs) or by reports from its own fault detector about failures/recoveries of neighboring servers. Messages received by the GMS can be either reports on membership events or RESOLVE and SET_FLAG requests originated by end-points.

When a server or an end-point is disconnected or fails, re-merges or recovers, a set of LMSs receives a membership notification and propagates it to interested end-points on their machines. This set is defined as following. Let $\mathcal{L}(G)$ be the set of LMSs that reside at the hosts where the members of a group G run. Let $\mathcal{L}'(G)$ be the set of LMSs that have members of G that request incremental membership notifications (have SET_ONLINE flag set). Note that $\mathcal{L}'(G)$ is always a subset of $\mathcal{L}(G)$. We denote the set of different groups that have members running under an address prefix *Addr* as $G(Addr)$. *Addr* can denote a domain, a host or a single end-point. If *Addr* is an address of an end-point, $G(Addr)$ in the set of groups of which *Addr* is a member. Hence the set $\mathcal{F}$ of LMSs that should receive the incremental membership notification will be

$$\mathcal{F} = \bigcup_{g \in G(Addr)} \mathcal{L}'(g)$$

[9]The neighboring servers can be either children, siblings or the parent of the GMS.

In the opposite direction, all end-points running on a re-merged LMS that have their SET_ONLINE flag set should receive incremental membership notifications from all LMSs belonging to $\mathcal{F}$. When an end-point issues a RESOLVE request for a group G, the request should be forwarded to the LMSs in $\mathcal{L}(G)$. When an end-point issues a JOIN, or LEAVE request for a group G, a notification should be forwarded only to LMSs in $\mathcal{L}'(G)$. When a domain fails, an incremental membership notification containing the address of this domain should be forwarded only to LMSs in $\mathcal{F}$.

In order to forward the membership notifications on a group G to the relevant set of LMSs only, each GMS maintains a knowledge of the directions in which LMSs from $\mathcal{L}(G)$ and $\mathcal{L}'(G)$ can be found. This knowledge is kept for every group G that has members in the GMS's domain subtree. This substantially reduces the network traffic caused by CONGRESS. Each GMS can maintain this knowledge at reasonable memory requirements as will be explained in the next subsection.

Due to the network delay and the asynchronous model that is assumed, this knowledge can be sometimes inaccurate. It is possible thus, that the set of LMSs that receive the notifications will be broader then necessary. CONGRESS guarantees however, that all the servers that belong to $\mathcal{L}(G)$ and $\mathcal{L}'(G)$, will eventually learn about the new membership.

5.2.2 GMS's Data Structures

The GMS's data structures maintain information needed by the GMS in order to know in which directions to forward different protocol messages. For each multicast group G we define the *control sub-tree* of G, $T(G)$, that serves for communicating control information relating to G (e.g. group name resolution requests, configuration changes etc.). $T(G)$ is defined as $T(G) = Graph(V(G), E(G))$ where $V(G)$ be the group of GMSs and LMSs that have members of G in their domain and $E(G)$ is the set of links connecting those servers in the CONGRESS hierarchy tree.

The GMSs do not maintain the full structure of $T(G)$. Each GMS has knowledge only of its local part of the graph. For each group G that has members in the GMS's domain, the GMS maintains a vector $Neighbors(G)$. This vector has one entry for each neighbor of the GMS, *i.e.* children, parent and siblings. Each entry can hold one of three values: a value of NONE means that no messages concerning the group G should be forwarded in the direction of the neighbor that this entry represents. A value of RESOLVE means that only RESOLVE requests concerning G are to be forwarded to that neighbor, since no end-points desiring incremental membership notifications reside in its domain. A value of ALL means that all messages concerning group G should be forwarded to that neighbor. In case of insufficient knowledge about the end-points in the domain of a neighbor its entry in the vector will be ALL as well. This is true, for example, at initialization time.

The vectors of a GMS form a Matrix, where each vector represents a row. The memory requirements for holding this matrix can be kept relatively small with the support of group scoping. The number of columns equals the number of neighbors. This number is limited by the degree of the CONGRESS hierarchy tree that corresponds the degree of the network domain hierarchy tree. The number of rows equals the number of groups that have members in the domain of the server holding this matrix. Although world-wide groups must be advertised at higher levels in the CONGRESS hierarchy, local groups, could be kept advertised only at lower-level domains by defining their scope to be as small as possible. It should be the user's interest to minimize the scope of a group in order to minimize delay of protocol messages concerning the group. Note that in case of a neighbor failure, the GMS will set the neighbor entry in all of the $Neighbors(g)$ vectors to be NONE.

The GMS also maintains a bit vector $Neighbors\_state$ which, like $Neighbors(G)$ has one entry per neighbor but which is not related to a specific group G. The corresponding entry will be marked CONNECTED if a neighboring server (*i.e.* parent, siblings or children) is operational from the GMS's point of view[10] and DISCONNECTED otherwise.

A third data structure held by the GMS is a queue of open resolve requests. This queue holds data on any resolve request received by the GMS that was forwarded and was not answered yet. The data consists of the *id* of the resolve request, the server from whom it was received and a *waiting-list* of neighbors. The *waiting-list* is the list of neighbors to which the request was forwarded and that did not reply yet. The GMS will use this information upon receiving a reply to this request, in order to forward the reply in the proper direction.

The GMS also holds an array of the ATM addresses of all its neighbors. A function maps each position in the vector $Neighbors\_state$ to an ATM address of the server.

6 Protocol

CONGRESS is an event driven protocol. CONGRESS servers receive messages that can come either from the network, initiated by other instances of the protocol, or from the fault detector module.

In this section we describe CONGRESS in greater detail. Subsection 6.1 gives an overview of the messages involved in the protocol. Subsection 6.2 describes in detail the data structures and variables used in the protocol. Subsection 6.3 describes the the LMS protocol and in Subsection 6.4 we describe the GMS protocol. Subsection 6.5 holds references to the various utility functions used in the protocol.

6.1 Protocol Messages

In this section we give the format of the messages that are used in all the parts of the protocol. Section 6.1.1 gives the format of the messages that are used for communication between the LMS and its GMS, and between the GMSs themselves, while Section 6.1.2 gives the format of the messages that is used for communication between an end-point and its LMS.

6.1.1 LMS/GMS messages

Figure 5 depicts the **registration_update** message format. On the bottom of this figure there is a table, which further explains the use of each field in the message.

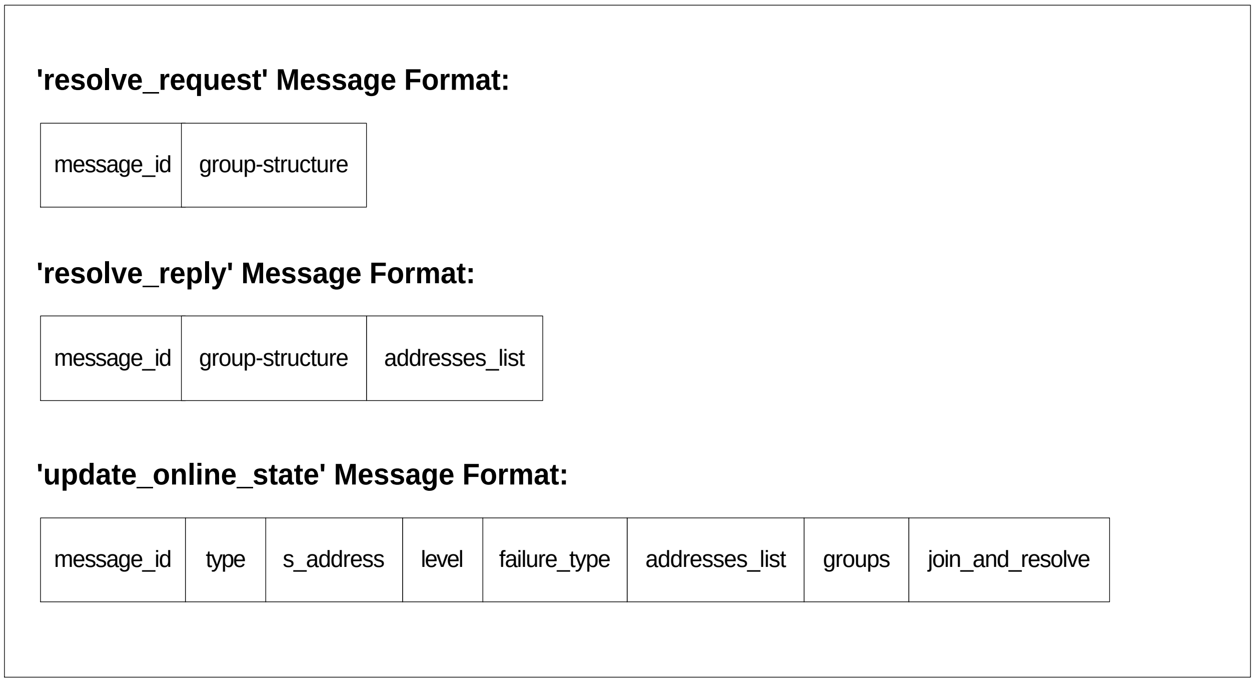
[10]As reported by the fault detector. Note that a GMS can consider a server faulty because of a broken link, although that server can remain active and communicate over other non-control links with the rest of the world.

Registration Update Message Format:

message_id	type	s_address	level	failure_type	addresses_list	groups	join_and_resolve

Field Type	**Description**
message_id	This Field is comprised of a sender ID and a local sequence number.
type	This field holds the message sub-type, which can be EP_JOIN or EP_LEAVE for the end-points' join or leave operations, or HD_FAILURE which stands for Host/Domain failures.
s_address	The server address of the Host/Domain that failed. This field is used only in the HD_FAILURE case.
level	This is the level of the CONGRESS server that failed. This field is used only in the HD_FAILURE case).
failure_type	This field can hold one of the two values: FILTER_IN or FILTER_OUT. For further explanations consult the description of the LMS Registration Message Handler.
addresses_list	This is the list of end point(s) that had joined (in case of EP_JOIN sub-type). In the current description of the protocol, user initiated join/leave operations would cause this list to contain only a single ATM end-point address.
groups	This field contains a list of group structures (group name and group scope) of the groups to which the message is related. In the EP_JOIN/EP_LEAVE case, this list contains a single group structure. In the HD_FAILURE sub-type case this list is used for noting which groups had members in the failed server domain (and thus to which directions to forward the message).
join_and_resolve	This field holds a boolean value (TRUE/FALSE). If this field is set to TRUE, then there are some special actions that the receiving server must perform. The use of this field is further explained in the "GMS Resolve Reply Handler" and the "GMS End-Point(s) Join Message Handler" .

Figure 5: Registration Update Message Format.


'resolve_request' Message Format:

message_id	group-structure

'resolve_reply' Message Format:

message_id	group-structure	addresses_list

'update_online_state' Message Format:

message_id	type	s_address	level	failure_type	addresses_list	groups	join_and_resolve

Figure 6: Other Protocol Messages Format.

Figure 6 depicts the format of the rest of the messages that the LMS and the GMS utilize in the protocol. It details the **resolve_request**, the **resolve_reply** and the **update_online_state** message formats.

6.1.2 End-Point requests/messages

Figure 7 depicts the format of the end-point requests, in terms of variables and fields. The request formats are compliant to the user interface given in section 4.4. Each function that is provided for a CONGRESS user has its variables packed in a message format. In the protocol, we refer to the variables in the user requests as fields in messages.

6.2 Protocol related Data Structures

In this section we list the data structures that the protocol requires. All these data structures are explained in greater detail in Sections 5.2.2 and 5.1.2.

- *my_level* : Used by the GMS.
  This variable holds the level of the GMS in the CONGRESS hierarchy;
- *Neighbors_state* : Used by the GMS.
  This variable is a vector with an entry for each neighbor of this GMS. The entries are marked either CONNECTED or DISCONNECTED depending on the state of the neighbors as reported by the fault detector module;

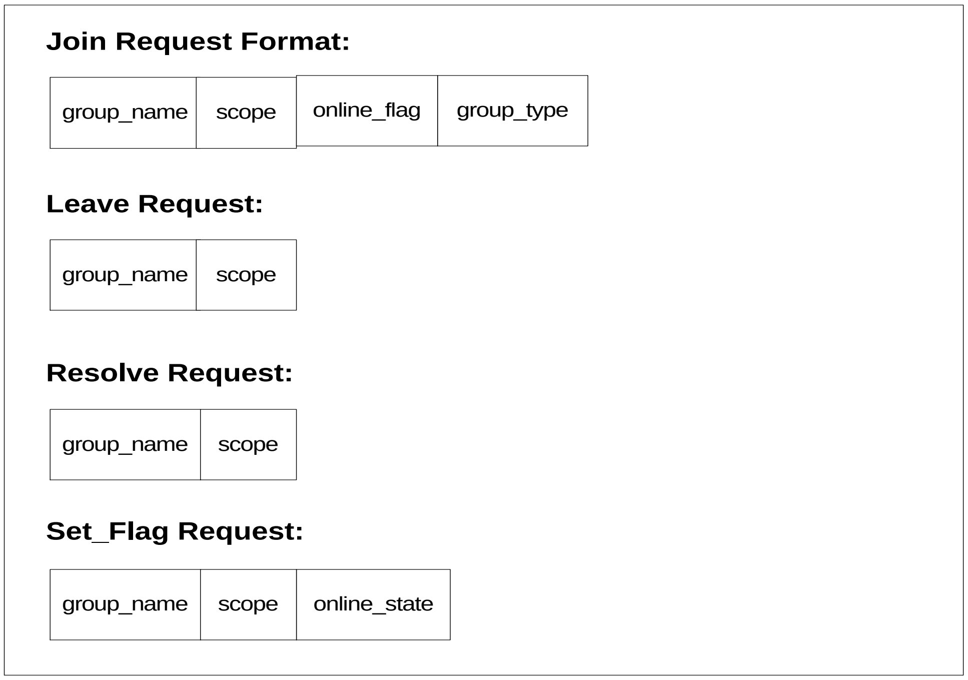


Figure 7: End-Point Requests Format.

- *Groups* : Used by the GMS.
  This is a matrix in which the GMS holds information about multicast groups. The rows represent multicast groups, and the columns represent neighboring CONGRESS servers. The entry (g, s) in *Groups* matrix may hold one of the following values:
  - NONE: This value means that no messages concerning the group g should be forwarded in the direction of the neighbor that this entry represents;
  - RESOLVE: This value means that only RESOLVE requests concerning g are to be forwarded to the neighbor that this entry represents. This means that no end-points desiring incremental membership notifications reside in this neighbor domain;
  - ALL: This value means that all messages concerning group g should be forwarded to the neighbor that this entry represents. In case of insufficient knowledge about the end-points in the domain of a neighbor its entry in the vector will be ALL as well.

  Although the scoping mechanism is used, this matrix can grow to relatively large sizes. This fact, and the fact that groups sometimes cease to exist (like when the last member issues a LEAVE request), call for a *garbage collection* mechanism to be incorporated into the protocol implementation. For simplicity's sake, we do not present a garbage collection mechanism in the protocol presented here;

- *Groups_info_table* : Used by the LMS.
  This is a vector of multicast groups names with an entry for each group that is

presented at the LMS's host. The entry relating to group g points to a list of end-points that are running on the LMS's host and belong to g. The size of this vector is subject to dynamic changes caused by processes that join or leave different groups or fail;

- *Open_resolve_requests_queue* : Used both by the GMS and the LMS.
  This is the queue in which each CONGRESS server holds the resolve requests that it had forwarded to neighboring CONGRESS servers, and that were not answered yet by **all** of these servers[11]. The LMS inserts the requests into this queue with a message *id* and a list of recipients (called *waiting_set*) that wait for a reply to the request. The GMS uses a similar message *id* to identify open requests in the queue, but it holds different information per open request:

  - *full_waiting_set*: This is the set of CONGRESS servers to which the GMS had **forwarded** the request. It is filled when the GMS invokes the ENQUEUE_RESOLVE_REQUEST utility function.
  - *waiting_set*: This is the set of CONGRESS servers to which the GMS had **forwarded** the request, and that did not replied with an answer yet [12]. First, it is filled when the GMS invokes the ENQUEUE_RESOLVE_REQUEST utility function, and then, for each reply that is received, the server that sent it is removed from the *waiting_set*;
  - *source*: This is the server that has sent the resolve request to the GMS, and to whom the GMS would return the aggregated result that was collected from the *waiting_set* of servers;

    It is possible that more then one CONGRESS server requests a resolve for a group g from the GMS before the resolve operation for g was finished. An optimization can be made in order to avoid sending multiple resolve requests for the same group. For each open request we would have to keep track of multiple sources. For each source we also save the request's message *id* to be returned along with the reply. This optimization is possible only when the two requesting servers are from the same "side" of the GMS - either both belong to the *children* of the GMS, or both belong to $parent \bigcup siblings$ of the GMS. For simplicity's sake we do not implement this optimization in the presented protocol;
  - *reply*: This is the aggregated reply that is collected from all the CONGRESS servers. Once a reply is received from all the CONGRESS servers in the *waiting_set*, the aggregated reply is sent to the request's *source*;
  - *resolve_join*: This is a flag, indicating whether the reply for this **resolve_request** should be forwarded back as a **join** message or as a **resolve_reply** message. The message is forwarded back to a neighbor as a JOIN message in case the GMS initiated the **resolve_request** due to that neighbor's reincarnation.

[11]If the server is an LMS, then it 'forwards' the request only to its local GMS, and thus it has to wait only to its reply.

[12]Note that this set is totally different from the *waiting_set* of the requests in the LMS queue. Here, *waiting_set* is the set of servers that should **reply** and not to **receive** a reply.

- *my_gms* : Used by the LMS.
  This variable holds the ATM address of the GMS of the cluster to which the LMS belongs;

- *gms_status* : Used by the LMS.
  This variable holds the status of the GMS of the cluster to which the LMS belongs. This variable can hold one of two values: OFF or ON;

6.3 LMS Protocol

In this section we will describe in greater detail the main event loop of the LMS (See Fig 8). There are three classes of events that the LMS handles. The first is the reception of a message from the GMS. The second is the interaction between the LMS and its local end-points. The third class of events contains two events: disconnection from the GMS and reconnection to the GMS. These events are generated by the underlying fault detector module and are reported as messages to the LMS.

When an LMS disconnects from its GMS, the LMS informs its local end-points that they are cut off from the CONGRESS services. In such a case any end-point running on the LMS belongs to a network component consisting only of one host: the host on which the disconnected LMS runs. In addition, the LMS changes its local variable reflecting the state of its GMS to be "OFF"

Upon reconnection to the GMS, the LMS changes the value of its local variable *gms_status* to be "ON". In addition, for each group represented at the LMS, the LMS notifies the rest of the group members that belong to the new (larger) network component of the LMS that the local members are again available[13].

The LMS propagates registration updates, triggered by local end-points issuing requests for leave or join or failures. It is also responsible for *delivery* of registration updates coming from the GMS. The LMS also forwards replies for the resolve request to the local end-points that expect it. As will be clarified in the following pseudo code, the LMS delivers exactly one reply for any request by an end-point for resolution of a group membership.

6.3.1 The LMS End-Point Join Request Handler

The code given in Figure 9 details the handler that is invoked when an LMS receives a **join** request r from a local end-point *ep*.

Note that r is a join request, and that it contains several fields (that are detailed in 6.1.2). We will use the requests fields in the next subsections only when necessary. A detailed description of these fields was given in 6.1.2.

The first thing that the LMS does when receiving a JOIN request with $r.group = g$ and a $r.scope = s$ is to check whether the invoking end-point belongs to s. If not, an error is delivered to the end-point. Next, the LMS checks for consistency between the actual group type of g and the group type that is specified in the request. In case the group type given is PERMANENT and the actual type of g is SPONTANEOUS, or vice-versa, then a proper error message is delivered to the end-point.

[13]This is equivalent to issuing a JOIN request by the LMS on behalf of the reconnected end-points.

```
procedure LMS-MAIN-LOOP():

    Loop forever {
        switch (event) {

/* LMS <-> END-POINT INTERACTION: */

            case received resolve request r from local end-point ep:
                HANDLE_RESOLVE_REQUEST(ep, r.group_name, r.scope);
            case received join request r from local end-point ep:
                LMS_JOIN_REQUEST_HANDLER(r, ep);
            case received set_flag request r from local end-point ep:
                UPDATE_END_POINT_ONLINE_STATE(ep, r.group_name, r.scope, online_state);
            case received leave request r from local end-point ep:
                LMS_LEAVE_REQUEST_HANDLER(r, ep);

/* LMS <-> GMS INTERACTION: */

            case received a registration update m from the GMS:
                HANDLE_REGISTRATION_UPDATE_MSG(m);
            case received a resolve request message m from the GMS :
                LMS_RESOLVE_MESSAGE_HANDLER(m);
            case received reply m on resolve request:
                LMS_RESOLVE_REPLY_HANDLER(m);

/* FAULT SUSPECTOR EVENTS: */

            case failure detection of local end-point ep:
                LMS_EP_FAILURE_HANDLER(ep);
            case failure suspicion of GMS:
                LMS_HANDLE_GMS_FAILURE();
            case reconnection to GMS:
                LMS_RECONNECT_GMS();

        } /* switch (event) */
    } /* Loop forever */
```

Figure 8: The LMS Main Event Loop.

```
procedure LMS_JOIN_REQUEST_HANDLER(r, ep):

    if (¬IN_ADDRESS_SCOPE($ep, r.scope$)) {
        ERROR($ep$ , NOT IN SCOPE);
        return;
    }
    if ($r.group\_type ==$ PERMANENT) and (not IS_PERMANENT($r.group\_name, r.scope$))
        ERROR($ep$ , NO SUCH PERMANENT GROUP);
    else if ($r.group\_type ==$ SPONTANEOUS) and IS_PERMANENT($r.group\_name, r.scope$)
        ERROR($ep$, GROUP ALREADY EXIST);
    else {
        $join\_msg =$ NEW(registration_update);
        $group =$ NEW(group_struct);
        $join\_msg.type =$ EP_JOIN;
        $join\_msg.addresses\_list = \{ep\}$;
        $group.group\_name = r.group\_name$;
        $group.scope = r.scope$;
        $join\_msg.groups = \{group\}$;
        $join\_msg.join\_and\_resolve =$ FALSE;
        /* inform local end-points of the new member */
        HANDLE_REGISTRATION_UPDATE_MSG($join\_msg$);
        UPDATE_GROUPS_INFO(ADD, $r.group\_name, r.scope, ep, r.online\_flag$);
        if ($gms\_status ==$ ON)
            SEND($\{my\_gms\}$ , $join\_msg$);
        UPDATE_END_POINT_ONLINE_STATE($ep$, $r.group\_name$, $r.scope$, $r.online\_flag$);
        if ($r.online\_flag ==$ SET_ONLINE)
            HANDLE_RESOLVE_REQUEST($ep, r.group\_name, r.scope$);
    }
```

Figure 9: The LMS End-Point Join Request Handler.

In case all the validity checks above were completed without an error, then the LMS takes the following steps:

- A **registration_update** message is formed. It contains a request subtype EP_JOIN, the *ep* identifier and the group structure (containing the group name g and the scope s). The flag *join_and_resolve* which is used in the GMS domain merging process is set to FALSE;
- The local end-points of g are informed about the joining end-point. This is done by using the registration message handler, which can be found in Figure 14 and explained in further detail in Subsection 6.3.6;
- The information about the new end-point is recorded in the local data structures using the routine UPDATE_GROUPS_INFO.
- If the GMS status is ON then the **join** message is sent to it;
- The *online_state* of the new end-point is checked, and if the new end-point changes the online state in the LMS regarding g, then a proper **update_online_state** message is

sent to the GMS. All this is done by calling to the UPDATE_END_POINT_ONLINE_STATE handler;

- If the end-point online state is SET_ONLINE, then a resolve operation is started for g, in order to inform the newly joined end-point about the group membership of the group that it had just joined.

6.3.2 The LMS End-Point Leave Request Handler

```
procedure LMS_LEAVE_REQUEST_HANDLER(r, ep):

  if (MEMBER(ep, r.group_name, r.scope)) {
      leave_msg = NEW(registration_update);
      group = NEW(group_struct);
      leave_msg.type = EP_LEAVE;
      leave_msg.addresses_list = {ep};
      group.group_name = r.group_name;
      group.scope = r.scope;
      leave_msg.groups = {group};
      UPDATE_END_POINT_ONLINE_STATE(ep, r.group_name, r.scope, RESET_ONLINE);
      UPDATE_GROUPS_INFO(DELETE, r.group_name, r.scope, ep, IGNORE);
      /* Update open resolve requests queue due to the ep leave */
      for each request ∈ Open_resolve_requests_queue do {
          if (ep ∈ request.waiting_set) {
              request.waiting_set = request.waiting_set \ {ep};
              if (request.waiting_set == {})
                 DEQUEUE_RESOLVE_REQUEST(request.id);
          }
      }
      DELIVERY(LOCAL_END_POINTS_OF_GROUP(r.group_name, r.scope, SET_ONLINE),
              EP_LEAVE, group, ep);
      if (gms_status == ON)
          SEND({my_gms} , leave_msg);
  }
```

Figure 10: The LMS End-Point Leave Request Handler.

The code given in Figure 10 details the handler that is invoked when an LMS receives a **leave** request r from a local end-point *ep* where *r.group* is denoted by g and *r.scope* is denoted by s.The handler code is executed only if *ep* is a member of the group g with the scope s.

Initially, the LMS forms a **registration_update** message. It contains the request subtype EP_LEAVE, the *ep* identifier and the group structure (containing the group name g and scope s). The LMS handles the online state of the end-point *ep* as if it was set to be RESET_ONLINE. This takes care of updating the GMS about the new state in the LMS regarding g. Note that the online state of the LMS does not necessarily change. The UPDATE_END_POINT_ONLINE_STATE handler takes care of all those details (See Section 6.3.7).

Next, the LMS updates its local data structure, which contains information about the local group membership. Then, the LMS updates all the open resolve requests that are in its queue. For each open resolve request that contains *ep* in its *waiting_set* field, the LMS removes the *ep* from this field, and if no other end point is waiting for the resolve reply, then the resolve request is dequeued.

The LMS informs the local end-points of the group that the *ep* has left, and then if the GMS state is ON, it sends it the **registration_update** message informing about the leave request.

6.3.3 The LMS Outgoing Resolve Request Handler

```
procedure HANDLE_RESOLVE_REQUEST(ep, g, s):

    if ($gms\_status$ == ON) {
        /* $g$ is the group name, and $s$ is its scope. */
        $r$ = FIND_OPEN_REQUESTS_FOR_GROUP($g, s$);
        /* Note that here $r$ contains only a single request structure */
        if ($r \neq$ NULL)
            $r.waiting\_set = r.waiting\_set \bigcup \{ep\}$;
        else {
            $group$ = NEW(group_struct);
            $group.group\_name = g$;
            $group.scope = s$;
            $m$ = NEW(resolve_request);
            $m.group = group$;
            /* queue the resolve request in the open resolves queue */
            ENQUEUE_RESOLVE_REQUEST($m.message\_id, group, \{ep\}, \{my\_gms\}$, RESOLVE);
            SEND($\{my\_gms\}$ , $m$);
        }
    }
    else {
        $group$ = NEW(group_struct);
        $local\text{-}end\text{-}points$ = LOCAL_END_POINTS_OF_GROUP($g, s$, NULL);
        $group.group\_name = g$;
        $group.group\_scope = s$;
        DELIVERY($ep$, absolute, $group$, $local\text{-}end\text{-}points$);
    }
```

Figure 11: The LMS Outgoing Resolve Request Handler.

Figure 11 details the handler that is invoked when an LMS receives a **resolve_request** from a local end-point *ep* for a group name *g* with the scope *s*.

In case the GMS status is OFF then the LMS immediately delivers the initiating end-point its local members of the requested group. Otherwise, the LMS checks whether it has an open resolve request for the same group in the queue. If there is already a queued open resolve request for *g*, then the LMS just adds the *id* of the requesting end-point to the *waiting_set* list of the queued resolve request. If there isn't a previous open resolve request for this group, then it forms a resolve request message m containing the requested group and scope, enqueues it and sends it to the GMS.

6.3.4 The LMS Incoming Resolve Message Handler

```
procedure LMS_RESOLVE_MESSAGE_HANDLER(m):

    reply_msg = NEW(resolve_reply);
    reply_msg.message_id = m.message_id;
    reply_msg.group = m.group;
    reply_msg.addresses_list =
            LOCAL_END_POINTS_OF_GROUP(m.group.group_name, m.group.scope, NULL);
    SEND({my_gms}, reply_msg);
```

Figure 12: The LMS Incoming Resolve Message Handler.

The code in Figure 12 details the handler that the LMS invokes when a resolve request message is received from the GMS. The invoking LMS retrieves the list of its local members of the group contained in the message, and puts it into a **resolve_reply** message. The *message_id* of the incoming message is copied into the reply message, so that the GMSs that had forwarded the request can forward the reply back until it is received by the request initiator.

6.3.5 The LMS Incoming Resolve Reply Message Handler

The code in Figure 13 details the handler that the LMS invokes when a resolve reply message is received from the GMS.

First, the LMS searches for the open resolve request matching the *message_id* field in the message m. In case no match is found in the open resolve request queue, the LMS discards the message. Note that a mismatch may occur in case the resolve request queue was either cleared due to a failure suspicion of the GMS [14], or if the request was dequeued. The request can be dequeued before the resolve operation is finished, if the end-point that has initiated it was the only end-point running on the sc lmss host that requested a resolve for that group, and if it failed or had issued a LEAVE request for that group.

In case a match was found, the LMS retrieves the list of end points that are waiting for the resolve reply. Then it adds its local members of the resolved group, to the resolve reply. Finally it delivers the full reply to the waiting end-points, and dequeues the resolve request which is no longer 'open'.

6.3.6 The LMS Registration Message Handler

The code given in Figure 14 details the handler the LMS invokes upon reception of a **registration_update** message. The action the LMS takes depends on the message sub-type:

[14]If the GMS did not fail, it might reply later to that resolve request, and the resolve request would no longer be in the queue

- EP_JOIN or EP_LEAVE: In this case, the **registration_update** message contains a list of end-points that have either joined or left a group g (that is also specified in the message). The LMS determines the list of local end-points, which are members of g and whose $online\_flag$ is SET_ONLINE. The LMS delivers to these local end-points the list of end-points specified in the message along with the message sub-type, so that each end-point can locally calculate the new group membership of the group g;
- HD_FAILURE: In this case, the **registration_update** message contains an address of a failed CONGRESS server, and its level in the CONGRESS hierarchy. The message also contains a list of groups which had members in the failed server's domain. The LMS calculates the domain prefix of the failed server address from its level. For each group g specified in the message, the LMS determines the list of local end-points that should receive a prompt message about the failure (*i.e.* are members of g and have the SET_ONLINE flag set). To each end points in the accumulated list a message is delivered denoting the failure of the host/domain, with the failure type, which can be either FILTER_IN or FILTER_OUT. These types of failures should be handled by the receiving end-point in the following way:
  - FILTER_IN: The end-point removes all the end-points **outside** of the given domain from all their groups membership. The removal is done based on the domain prefix that was derived from the message;
  - FILTER_OUT: The end-point removes all the end-points **belonging** to the failed domain from all their groups. The removal is done based on the domain prefix that was derived from the message.

6.3.7 The LMS Online State Update Handler

In Figure 15 we give the procedure that handles the changes of the *online_state* of an end-point *ep*. This change can happen due to either a user request such as JOIN, LEAVE or SET_FLAG, or an *ep* failure. The procedure receives an end-point address, a group name and its scope, and an *online_state* flag. It sets the online flag of the end-point to the value of *online_state*, and reports the change to the GMS in case this change has affected the LMS online updates flag for this group. This function is also called when an end-point fails or leaves a group.

6.3.8 The LMS End-Point Failure Handler

The code given in Figure 16 details the handler the LMS invokes upon receiving a message from the fault detector module, regarding a suspicion of a failure of a local end-point *ep*.

First, the invoking LMS forms a **registration_update** message, with the sub-type of EP_LEAVE. The address list field of the message is filled by the end-point address *ep*, and the groups field of the message is filled with the groups of which the failed end-point *ep* was a member. In case the GMS status is ON, then the message is sent to it.

Next, the invoking LMS updates the open resolve queue; The failed end-point *ep* is removed from the list of any open resolve request that has *ep* is in its waiting end-points list. If after this removal the waiting end-points list of a request becomes empty, then the whole resolve request is removed from the queue.

Finally, for each group g of which *ep* was a member, the invoking LMS does the following:

- The online state handler is invoked, as if *ep* initiated a SET_FLAG request for g with the *online_flag* set to RESET_ONLINE. This ensures that in case the end-point failure caused a change in the LMS *online_state* for group g, then the proper **update_online_state** message will be send to the GMS;
- The end-point *ep* is removed from the local data structures regarding g;
- The local end-points the are members of the group g receive a message reporting that the end-point *ep* has left the group.

6.3.9 The LMS Handler Of The GMS Failure

The code given in Figure 17 details the handler the LMS invokes upon receiving a message from the fault detector module, regarding a failure suspicion in the GMS.

First, the LMS sets its local variable *gms_status* to OFF. Then it forms a list containing the *id*'s of all its local end-points that have their *online_flag* set to SET_ONLINE.

Next, to this list of end-points the LMS delivers a message of type FILTER_IN, that contains the LMS's address. By this message the receiving end-points are notified that only the end-points that belong to the domain of the received address (*i.e.* the LMS address) should remain in any of their group memberships.

Finally, the LMS forms an empty **resolve_reply** message. For each open request in its open-request queue it does the following:

- It determines the set of local end-points that is waiting for the resolve reply;
- It fills the group from the open resolve request into the group field of the resolve reply message;
- It fills the resolve reply field with the list of its local end-points, that are members of the resolved group;
- It delivers the resolve reply message to the waiting end-points;
- It dequeues the open resolve request.

Note that by the end of the above process, the open resolve requests queue is empty and will remain so, until the GMS will be reconnected.

6.3.10 The LMS Reconnection To The GMS Handler

The code given in Figure 18 details the handler the LMS invokes upon receiving a message from the fault detector module, regarding a re-incarnation or re-connection of the GMS.

First, the invoking LMS sets the *gms_status* to ON. Then for each group g that is registered on the LMS, it does the following:

- Determine the local members of g, and put them in *memb_list*;
- Form a **registration_update** message, and fill in all its details;
- Send the registration update message with the sub-type EP_JOIN, and thus cause the local end-points re-join to the global group;
- Form an **update_online_state**, and fill in the LMS *online_state* regarding the group g;
- Send the online update message, and thus setting the LMS's online state at the GMS data structures.
- If there are any local end-points that have their *online_flag* set to SET_ONLINE (regarding to group g), then form a **resolve** message for the group g, set the waiting end-point list to the list of those end-points, enqueue this resolve request in the queue and send this resolve message to the GMS. This would take care of informing the interested end-points of their new global group membership, after the reconnection to the GMS.

6.4 GMS Protocol

In this section we will describe in greater detail the main event loop of the GMS. There are two kinds of events that a GMS handles. The first is the reception of a message from one of its neighbors. This message can be received from either an LMS that is in its cluster, or from a neighbor GMS. Note that a neighbor GMS can be either the parent of the GMS, one of its children, or one of its sibling. The second kind of events is the reception of a report from the fault detector module, which can be either the suspicion of a neighbor to be faulty, or the detection of reincarnation of a neighbor that was suspected to be faulty.

One important implementation issue must be dealt here: As a consequence of the CONGRESS hierarchy it is obvious that there can be more than one GMS running on the same host. This can be done in one of two ways: The first is by executing a single process that performs the work of several GMSs. This process must be capable to identify the level, and thus the GMS to which each message/event relates, and handle it properly. The second, is to use multiple processes for multiple GMSs running on the same host. Since the second way simplifies the protocol, we chose to write the protocol with respect to it. Hence, when we use the notion n in order to refer to some neighbor of the GMS, it can be either another GMS running on the same host, or another GMS running on another host. In order to distinguish between the GMSs, n would contain the host address on which the neighboring GMS runs, and an additional identifier, such as the VCI in ATM, etc.

The GMS process resolve requests and registration update messages. Note that if a GMS receives a resolve request from one of its neighbors, it sends only one reply for this specific resolve request. This is guaranteed by the way the GMS handles this request. It first forwards the request to the subset of its neighbors from which it has to collect group membership information. Then, it collects all the replies for this resolve request from these neighbors, and only when all of them have replied, it forwards the aggregated reply to the request initiator.

When a GMS detects a neighbor that has reconnected (by the fault detector module), it starts a re-merge operation. First it determines all the groups that are registered in a scope greater or equal to the level of the reconnected neighbor. Then it issues a resolve request to all of its children for each of these groups. It enqueues these resolves requests and mark them as RESOLVE_JOIN requests. Then for each of these groups it sets the entry in the *Groups* matrix in the position of the reconnected neighbor to be ALL. Upon the complete reception of the replies to these resolve requests, the GMS forwards the list of end-point addresses as if those were JOIN requests by those end points. The target neighbor will be the re-connected neighbor. Note that in case that a GMS was disconnected from the rest of the CONGRESS hierarchy, and now it has re-connected, the above process of re-merging would start for each of the neighbors that are now connected again. This can be easily optimized, by forwarding the resolve requests replies as JOIN messages to **all** the new neighbors that were detected, and thus preventing redundant resolve operations. In order to simplify the presentation of the protocol this optimization is not presented in this paper.

6.4.1 The GMS Fault Suspicion Handler

Figure 20 shows the GMS's handler of a fault suspicion concerning a neighbor CONGRESS server as reported by the fault detector module. The invocation of this handler is triggered by an asynchronous event occuring at a specific *level* in the CONGRESS hierarchy. The function accepts a single argument: n - a neighbor's unique identifier (*e.g.* , ATM address and a VCI). The operation of the handler proceeds as follows: Initially, the invoking GMS identifies all the groups about which the presumably failed neighbor had information, and puts them in the *affected_groups* variable. This is done using the local *Groups* matrix. Next, the GMS marks the failed neighbor as DISCONNECTED. Afterwards the GMS builds the target group $\mathcal{TG}$ that includes neighbors to which a **registration_update** message about the failed neighbor should be sent. Depending on the hierarchy relationship between the invoking GMS and the failed neighbor, the $failure\_type$ field of the **registration_update** message may hold either:

- FILTER_IN: this means that the invoking GMS and all its siblings become disconnected from the rest of the CONGRESS hierarchy. This happens when the GMSL of the invoking GMS fails. The end-points residing in the domain of the failed GMSL should remove all the end-points outside of this domain from all the affected groups. The removal is done based on the domain prefix that is easily obtained from the address of the failed neighbor and its level. The $\mathcal{TG}$ of the **registration_update** message of the type FILTER_IN includes only the children that are currently connected and have some information on at least one of the *affected_groups*;

- FILTER_OUT: this is used when the failed server is either a *child* or a *sibling* of the invoking GMS. In this case all the end-points in the failed server domain are disconnected from the rest of the CONGRESS hierarchy. Hence, these end-points should be removed from all the affected groups' memberships throughout the network component of the invoking GMS. If the failed server was a *sibling* of the invoking GMS then the $\mathcal{TG}$ of the **registration_update** message of the type FILTER_OUT includes the children of the invoking GMS only. Otherwise, the message is sent to all the neighbors of the invoking server. In both cases messages are sent only to the servers that are currently connected and have some information on at least one of the *affected_groups*.

In case the failed server was the last one that should have replied on some of the opened resolve requests, then these requests become accomplished and are handled accordingly.

6.4.2 The GMS Server Reincarnation Handler

Figure 21 presents the GMS's handler of a neighbor CONGRESS server (*i.e.* another GMS or LMS) reincarnation. This function accepts a single argument: n - a neighbor's unique identifier. The operation of the handler proceeds as following. First, the GMS constructs a *Merging Groups* set $\mathcal{MG}$. $\mathcal{MG}$ includes all the multicast groups from the GMS's *Groups* matrix such that their scope embraces the recovered entity n. Next, for each group g from $\mathcal{MG}$, the GMS builds the Target Group set $\mathcal{TG}$ that holds all the GMS's children that are currently marked as holding information about the multicast group g. Following that, the GMS forms a **resolve_request** message m for group g and enqueues it as a RESOLVE_JOIN message in its local *Open_resolve_requests_queue*. The enqueued message is of type RESOLVE_JOIN in order to ensure that once the reply for m is formulated, it will be forwarded to the recovered entity as a **join** message. Note that forwarding a **join** message to the recovered entity is crucial for the protocol's correctness. Indeed, in order for group's members to be merged, mutual join messages must be exchanged. A **resolve_reply** alone is meaningless in this case, because it is not expected at the recovered server.

6.4.3 The GMS Resolve Request Handler

Figure 22 details the handler that is invoked when a GMS receives a **resolve_request** message for a group address. The first step the invoking GMS takes is to determine whether it has some information about the requested group at all. If the requested group is not registered in the GMS's *Groups* matrix, then the invoking GMS forwards the resolve request to all the relevant neighbors. The set of relevant neighbors is determined in the following manner: If the resolve request was received by the invoking GMS from its parent or one of its siblings, then the relevant set to which the message should be forwarded consists of its currently connected and operational children. Otherwise the set of relevant neighbors contains all currently connected and operational siblings. The parent may be added to this set in case the scope of the requested group spans its level.

Next, the invoking GMS enqueues the resolve message in the open resolve requests queue (in order to be able to correlate it with the responses received for it later on) and then forwards the message. If the requested group is already registered at the invoking GMS *Groups* matrix, then the whole process proceeds exactly as described above with the only exception, that the resolve message is not forwarded to any of the neighbors that are marked as NONE.

6.4.4 The GMS Resolve Reply Handler

Figure 23 details the handler that is invoked when a GMS receives a **resolve_reply** message on a previously issued resolve request for a specific group address. Basically, upon reception of the resolve reply, the invoking GMS correlates it with one of the resolve requests from its open resolve request queue, combines this reply with the previously received replies corresponding to the same request, and, when the full information about the requested group is obtained, sends it back to the request originator. There are two special cases, however, that demand special treatment.

First, it is possible that the reply cannot be matched with any of the requests queued in the open request queue. This may happen if the originator of the reply was recently presumed failed by the invoking GMS, and, thus all opened resolve requests concerned with the presumably failed server were answered and dequeued without waiting for a reply from the failed server. If no matching between the *id* of the reply message and the *id* of a queued resolve requests is found, the invoking GMS discards the reply message without further processing.

The second case refers to treatment of resolve replies which correlate with a resolve request of type RESOLVE_JOIN. As described in Subsection 6.4.2, such a resolve request is not initiated by any end-point, but is triggered by the detection of a neighbor's reincarnation. In contrast to a regular resolve request, there is no return address to which the reply on such a resolve request should be sent. When a reply to this request is received, it is forwarded to the recovered server as a JOIN message, so that the recovered GMS will be able to integrate smoothly (from the point of view of the recovered GMS, it is the rest of the world that had recovered from a failure and now asks to join back).

Note also, that there is an additional flag on any JOIN message that is initiated by the GMS when receiving a reply to a RESOLVE_JOIN request. This flag is called *join_and_resolve* and is set to TRUE if the reincarnated neighbor was a parent of the invoking GMS, and FALSE otherwise. This flag is necessary in order to trigger the resolve and join requests at all the levels in the CONGRESS hierarchy with respect to the scope of each merging group in order that the process of groups merging would not stop at a specific level. This is further explained in the Subsection 6.4.7.

6.4.5 The GMS Online State Message Handler

The code given in Figure 24 details the handler that is invoked when a GMS receives an **update_online_state** message m for a group g from a neighbor n.

First thing the invoking GMS does is to check whether the group g is in its *Groups* matrix. If not, nothing is done. Otherwise, the invoking GMS determines the current online state flags of its neighboring servers (children, siblings and parent) in its *Groups* matrix. Then, it update its *Groups* matrix, *i.e.* it puts the new online state value from the message, in the neighbor n entry for the group g row. Next it determines again the online state of its children and of its parent and siblings. If the online state of any of its neighboring servers was changed due to this update, then the message m should be forwarded. Otherwise, the invoking GMS finishes the handling of this message.

If the message m is to be forwarded, it is forwarded to a set of neighbors $\mathcal{TG}$ that is determined in the following manner: if the online state of the children was changed (hence n has to be one of the children) then $\mathcal{TG}$ is the group of siblings and the parent, with the exclusion of all the disconnected servers, and the servers that do not need any information about the multicast group g. If the online state of either the parent or the siblings was changed (hence n was one of them), then $\mathcal{TG}$ is the group of children, after similar filtering. Note that, if the level of the invoking GMS is 0, then $\mathcal{TG}$ is a subset of LMSs residing in the cluster of the invoking GMS. In this case, no message is forwarded, since the LMSs do not maintain any information about theirs GMS's online state.

6.4.6 The GMS Query For A Neighbor's Groups

In Figure 25 we give a simple function that the GMS uses in order to retrieve all the groups in which a neighbor n has interest, *i.e.* the groups which have members in the neighbor's domain. This information is obtained from the *Groups* matrix.

6.4.7 The GMS End-Point(s) Join Message Handler

The code given in Figure 26 details the handler that is invoked when a GMS receives a **registration_update** message of the sub-type EP_JOIN for a group g from a neighbor n. Several subtle points that require special treatment are described here.

CONGRESS guarantees to report membership events reflecting the chronological ordering of their occurrence. Thus the order between the resolve replies and join messages must be preserved. However, resolve replies progress in a much slower fashion across the CONGRESS hierarchy than EP_JOIN messages because resolve replies are delayed by GMSs that need to accumulate replies from other neighbors before forwarding them further.

For example, suppose that a **resolve_reply** is issued by an LMS as a reaction to a **resolve_request** for a group g initiated by an end-point *ep*. Suppose further that an EP_JOIN message for the same group g is initiated by the same LMS because a new end-point ep' appeared in its domain, and that this message follows the **resolve_reply** message. If the **resolve_reply** message will reach *ep* **after**[15] the EP_JOIN message will reach it, then, since the **resolve_reply** message does not include ep' in its membership, *ep* will obtain an 'old' membership of g. Furthermore, if *ep* has its online flag set to RESET_ONLINE or if no further changes occur in the g's group membership, it has no chance to learn about the new end-point unless another resolve request is initiated.

This scenario implies that wasteful delay might be imposed on join messages because of preceding RESOLVE MESSAGES. For this, we propose the following solution. When an EP_JOIN message for an end-point *ep* is received by a GMS, the queue of the opened resolve requests is checked. If there are opened resolve requests targeted to the same group g as the received EP_JOIN message, then *ep* is added to the reply list of end-points for this **resolve** message and the EP_JOIN may be immediately forwarded. Before forwarding the message, the GMS updates its *Groups* matrix, according to the group to which the message relates. Then the message is forwarded to the target set $\mathcal{TG}$ that is determined similarly to the way described in the sections above. In this way, although we alter ordering of messages, membership events are still reported in the order that reflects their chronological order.

A second issue that should be mentioned is the handling of EP_JOIN messages that have their *join_and_resolve* flag set. Such messages are initiated during the domain merging process by GMSs through the server reincarnation handler (See Section 6.4.2). In such cases, two things must be done:

[15]Due to a delay in one of the GMSs on its way back to the requesting *ep*.

- Information about the end-points from the message's address list must be propagated to all the relevant servers (*i.e.* those who have members of the group g in their domain);
- the GMS must determine the set of end-points that are members of group g and reside in its domain, and then should send a JOIN message containing them, back to n.

Note that if the neighbor n from which the EP_JOIN message (with its *join_and_resolve* field set to TRUE) was received is a child of the invoking GMS, then the invoking GMS is **not** supposed to do any actions regarding its own domain. This is because these actions were already performed at the lower levels in the CONGRESS hierarchy. Thus, in this case, the invoking GMS should only forward this JOIN message, with its *join_and_resolve* field set to TRUE, to the target set $\mathcal{TG}$ which is calculated as described earlier.

Note also that if n is a sibling of the invoking GMS then the *resolve_and_join* field of the message will become FALSE before forwarding it to the proper target set (which will be the children of the invoking GMS, or a subset of them). Thus, the receiving servers will handle this message as a regular **resolve_requests** message. This is done because the receiving servers do not need to repeat the same process for their domains.

During the server reincarnation process, the parent of a CONGRESS server that invokes the HANDLE_SERVER_REINCARNATION handler, is **passive** and does not initiate any protocol messages caused by its child reincarnation. Thus the CONGRESS server invoking the handler in Figure 26 will never receive EP_JOIN message with the *resolve_and_join* flag set to TRUE. This is why there is no handling of such a case in Figure 26.

6.4.8 The GMS End-Point Leave Message Handler

The code given in Figure 27 details the handler that is invoked when a GMS receives a **registration_update** message of the sub-type EP_LEAVE for a group g from a neighbor n.

Most of the operations that the GMS should do are the same as in the end-point(s) join handler (See Section 6.4.7). The GMS determines the target set of neighbors to whom it will forward the **registration_update** message, and then it forwards it.

There is one non-trivial point in this handler: In case there is an open resolve request for the group g in the open resolve requests queue, then the GMS removes the end-point that is in the address list field of the message from the resolve reply that is in the queue. This is done in order to prevent a resolve reply for a group from being received at its initiator, while containing an end-point that had issued a leave request **after** the resolve reply was sent from its LMS. This situation is exactly the same as in 6.4.7, with the only exception that in 6.4.7 the problem referred to EP_JOIN messages, and here we deal with EP_LEAVE messages.

6.4.9 The GMS Host/Domain Failure Message Handler

The code given in Figure 28 details the handler that is invoked when a GMS receives a **registration_update** message m of the sub-type HD_FAILURE for a group g from a neighbor n.

First, the invoking GMS checks whether the address of the failed host/domain in the message m is of one of its neighbors. If so, and if this neighbor is already marked as DISCONNECTED, then nothing is done.

Otherwise the invoking GMS determines the target set of neighbors to which the message m will be forward. It does so in a similar way to the one described in the previous subsections. In case the failed server in the message is a neighbor of the invoking GMS, then it adds the groups which had members in the domain of the failed neighbor, to the groups listed in the message. This is done in order to ensure that all the servers that had members of any of these groups in their domains (and have an online flag for any of these groups set to SET_ONLINE), would receive the message about its failure. Since the target neighbors set is determined using the groups of the failed server which are in the message, it is important to keep the *groups* field in the message as updated as possible. Thus, if the invoking GMS is forwarding the message to its children, and the failed server is one of its siblings, then it adds all the groups in which the failed server had interest (from the invoking GMS point of view) to m. This will ensure that all the relevant children will be in the target set.

6.5 Utility Functions

In this subsection we present a few basic functions that facilitate the protocol. The functions are divided into three sets. The first set contains functions that are used only by the GMS. The second set contains functions that are used only by the LMS, and the third set contains functions that are used by both the GMS and the LMS.

6.5.1 GMS Utility Functions

In the functions below, the argument l holds the level of the invoking CONGRESS server in CONGRESS hierarchy. The GMS's at the *clusters* - at the lowest level belong to *level* 0, the GMSLs of *level* 0 belong to *level* 1 **and** 0, etc.

- SIBLINGS(l): This function receives a level l in the hierarchy, and returns a set of addresses of the GMS' siblings in that level;
- PARENT(l): This function returns the address of the parent of the invoking GMS in level l;
- CHILDREN(l): This function returns a set of the GMS's children addresses. If the calling GMS is a GMSL of level $l-1$, which means that it is a GMS **member at level** l (this GMS can be also a GMSL for level l) CHILDREN(l) would return its children set of level $l-1$. If the GMS is not the GMSL for level $l-1$, this function returns an empty set;
- NEIGHBORS_SET(l): This function returns a set of addresses which comprises all the neighbors of the GMS in level l. The neighbors set is the union of the SIBLINGS, the CHILDREN and the PARENT sets of the GMS in level l;
- NEIGHBOR_STATE($neighbor$): This functions receives a *neighbor* id and returns its state which can be either CONNECTED or DISCONNECTED depending on its value in the $Neighbors\_state$ vector;
- SET_NEIGHBOR_STATE($neighbor, state$): This functions sets the proper entry for *neighbor* in $Neighbors\_state$ vector to be equal to *state*. The *state* variable can be either CONNECTED or DISCONNECTED;

- SEND($target\_set, msg$): This function receives a set of addresses and a message. It sends the message to all the addresses in this set. In case the *target_set* is empty, it does nothing;

- INDEX_BY_GROUP(g): This function receives a group structure g (containing the group name and its scope) and returns the line number - index of the group in the *Groups* matrix. This is actually the index in the *Groups* matrix of the vector that holds the GMS information about the group g. If there is no group g in the *Groups* matrix then the return value is -1;

- INSERT_NEW_GROUP(g): This function receives a group structure g (containing the group name and its scope) and inserts a new row into the *Groups* matrix for the group g. Note that **all** the entries of this row are initialized to ALL, since the GMS does not have more accurate data at the moment of the insertion;

- SCOPE_TO_LEVEL($scope$): This function receives a scope in the CONGRESS hierarchy, and returns the matching level l for the scope. This means that the scope *scope* will not include any CONGRESS servers that are at level l' s.t $l' > l$;

- GET_ONLINE_STATE($group, neighbor\_set$): This function receives a group structure (containing group name and its scope) and a neighbor set. The function returns ALL in case one of the *neighbor_set* CONGRESS servers has its online state set to ALL in the *Groups* matrix, and RESOLVE otherwise. In case the *group* is not represented in the *Groups* matrix, a value of -1 is returned.

6.5.2 LMS Utility Functions

- IS_PERMANENT($group\_name, scope$): This function receives a group name and its scope, and returns a boolean value: $True$, if the group (comprised of the pair $< group\_name, scope >$) is a permanent group, and $False$ otherwise;

- UPDATE_GROUPS_INFO($op\_type, group\_name, scope, ep, online\_state$): The operation type *op_type* can be either ADD, DELETE or SET_ONLINE. If *op_type* is ADD then the LMS will add the end-point *ep* to the $Groups\_info\_table$ in the proper entry for group *group_name* with all the additional information (online state, etc). If the group is new to the $Groups\_info\_table$, then a new entry would be made in the table. If the operation type is DELETE, then the proper deletion action would be taken. If the operation type is SET_ONLINE, then this function will set the online state of the end point to be *online_state*;

- MEMBER($ep, group\_name, scope$): This function checks whether a local end-point *ep* is a member of the group *group_name* with the scope *scope*. It does so by consulting the $Groups\_info\_table$. The function returns $True$ or $False$ respectively;

- LOCAL_END_POINTS_OF_GROUP($group\_name, scope, online\_state$): This function consults the $Groups\_info\_table$ and retrieves all the end-points that resides on the LMS's host and that fulfill the following properties: they are members of group *group_name* of the scope *scope* and their online flag is equal to *online_state*. The $online_state$ can be either SET_ONLINE, RESET_ONLINE or NULL. If the *online_state* is NULL, then the online flag of the end-points is not checked, and all the local members are collected, regardless of their online flag;

- GROUPS_OF_LOCAL_END_POINT(*ep*): This function receives an end-point *ep* address, and returns a list containing all the groups (each is a structure comprised of name and scope) of which this end point is a member of. This function uses only local information - the *Groups_info_table*;

- GROUPS_OF_LMS(): This function returns a list containing all the groups (names and scopes) that have members residing on the LMS's host;

- DELIVERY(*end-points*, **notification_type**, [*group*], *address(s) list*): This function delivers a notification with a given type for all the end-points in the set *end-points*. Depending on the notification type there might be a group to which the message relates (in the case of EP_JOIN for example). Delivery is done locally on the host of the LMS via inter process communication;

- ERROR(*end-points*, *m*): This function delivers the error message *m* to all the end-points in the set *end-points*.

6.5.3 General Functions

- IN_ADDRESS_SCOPE($ep, scope$): This function checks whether an end-point identifier (address) *ep* is in the scope *scope*. It actually checks if *scope* is a prefix of the end-point address. If it is, then the end-point is in the scope, and the return value is $True$, otherwise it returns $False$. Note that in our implementation the scope is identified by an ATM address prefix. Thus, the scope *s* consists of all the machines which have an ATM address with a prefix *s*. In other network technologies, such as frame-relay, another mechanism may be used to achieve scoping, and thus this function will be implemented and called with different argument types;

- MAKE_PREFIX($addr, l$): This function receives an ATM address of a CONGRESS server, and a level *l* in the CONGRESS hierarchy. It calculate the proper address prefix of *addr* corresponding to the level *l*. Note that as *l* increases, the prefix becomes shorter, and vice versa;

- ENQUEUE_RESOLVE_REQUEST($request\_id, group, source, request\_targets, type$): The *source* argument can be either a set of CONGRESS servers or end-points to which the **reply** to this request should be forwarded. The *request_targets* is a set of CONGRESS servers to which this **request** should be forwarded. In the current description of the protocol, the GMS uses only a single CONGRESS server, but the LMS can use this field to hold a list of end-points. This function puts this information in the *Open_resolve_requests_queue*, with the *request_id* as a key, to be retrieved upon request. The *Open_resolve_requests_queue* is described in further details in Section 6.2);

- DEQUEUE_RESOLVE_REQUEST(*request_id*): This function removes the request structure *r* with $r.id == request\_id$ from the *Open_resolve_requests_queue*;

- FIND_OPEN_REQUEST_BY_ID(*request_id*): This function finds and returns the request structure with $id == request\_id$ that is queued in *Open_resolve_requests_queue*;

- FIND_OPEN_REQUESTS_FOR_GROUP(*group_name*, *scope*): This function finds and returns a list of request structures r_i with each $r_i.group\_name == group\_name$ and $r_i.scope == scope$ that is queued in *Open_resolve_requests_queue*. In case no match is found, then a NULL value is returned. Note that the GMS may hold more than one open resolve request for a group. The LMS protocol, however, is more optimized, and holds only a single open request for a group, with the possibility for multiple reply targets;

- NEW(*type*): This function receive a data type, and returns an instance of that data type (variable).

7 Conclusion and Future Work

We have presented a CONnection-oriented Group-address RESolution Service for the native ATM environment. CONGRESS uses a logical name space for group addressing and enables maintenance of dynamic multicast groups. The protocol is sensitive to network and host failures. It exploits the network's hierarchical addressing scheme to support a world-wide scalability and scoping.

CONGRESS itself is not fully fault tolerant. If a GMS fails, all end-points that reside in its domain are cut off from CONGRESS services outside that domain. The protocol could be made much more robust by using replicated GMSs to serve each domain and by running an election algorithm to choose a new GMSL if a GMSL of a domain fails. When this will be implemented, receiving a notification on a partition from a domain could also indicate with a higher probability, that there is indeed a partition from that domain and it is not merely a result of a server crash. We see this as one of the issues where CONGRESS can be improved.

Several enhancements could be made to make the registration and scoping mechanism more flexible to various application demands. Although the scoping mechanism reduces the chance that two end-points would want to initiate groups of the same name, the possibility certainly exists. This could be a burden for applications that require unique groups and would have to leave and form a new group in order to operate. A parameter could be added to the JOIN request that states that the requested group name must be unique. A two-phase commit protocol can be used to lock the name until the initiator of the group is assured that no other end-points are members of the group.

Many applications have security requirements concerning multicast groups. For example, it might be necessary to limit the membership of a multicast group. *Closed* groups could be supported by CONGRESS for this purpose. Membership of a closed group is predefined by its initiator as a list of end-points that may ever join the group. JOIN and RESOLVE requests for such a group would be accepted only from end-points appearing on that list.

On the opposite side of the scale are groups for which there is no indication where the members will be located. Instead of specifying a group with a world-wide scope and waste bandwidth, "open" groups for which no scope is specified could be desirable. The scopes for such groups would be determined by the current membership of the group. If the current membership of the group spans a small domain, the current scope of this group would be that domain. Once a distant end-point desires to join the group, the scope of the group could be dynamically expanded to include that end-point. An additional research is necessary in order to find how to map such groups into a most effective representation.

The protocol presented in Section 6 assumes that an LMS runs on each host participating in CONGRESS. As mentioned earlier, a support for remote LMS' clients could be given. This would limit the ability to support process-level failure-detection using the LMSs but would enable the use of CONGRESS in cases where otherwise would be impossible (such as with PCs running DOS) or wasteful (when only a few processes per machine use CONGRESS).

The sensitivity of CONGRESS to host failures and network partitions might prove as a handy tool for a class of fault-tolerant reliable multicast packages such as Transis [2], Isis [10, 11] and Horus [24] when operating in an ATM environment. These packages make use of failure detectors in order to provide strong group semantics to applications using them. Members in a group need to reach *agreement* on membership in order to make conclusions about common knowledge. Instead of directly using failure detectors, which would report failures separately, the aggregate membership information supplied by CONGRESS could be used by group members to reach agreement on membership faster. This area is still to be investigated.

Acknowledgments

We are thankful to Prof. Israel Cidon for his helpful comments and suggestions. Special thanks to Idit Keidar, Gadi Shamir and the other members of the Transis project for their valuable remarks.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 647*, pages 292–312, November 1992.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992. The full version of this paper is available as TR CS91-13, Dept. of Comp. Sci., the Hebrew University of Jerusalem http://www.cs.huji.ac.il/labs/transis/transis.html.

[3] G. Armitage. *Internet Draft: Support for Multicast over Uni 3.0/3.1 based ATM Networks.* Bellcore, February 1996. Work In Progress.

[4] ATM Forum. *ATM User Network Interface (UNI) Specification Version 3.1.* Prentice Hall, Englewood Cliffs, NJ, June 1995. ISBN 0-13-393828-X.

[5] The ATM Forum Technical Committee. *ATM User-Network Interface (UNI) Signalling Specification Version 4.0, af-sig-0061.000*, July 1996.

[6] The ATM Forum Technical Committee. *ATM User-Network Interface (UNI) Signalling Specification Version 4.0, af-sig-0061.000*, July 1996. Section 7, pages 55-58.

[7] The ATM Forum Technical Committee. *Native ATM services: Semantic Descrption Version 1.0, af-saa-0048.000*, February 1996.

[8] The ATM Forum Technical Committee. *Private Network-Network Interface Specification Version 1.0 (PNNI 1.0), af-pnni-0055.000*, March 1996.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 7. Addison Wesley, 1987.

[10] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Annual ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[11] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transaction on Computer Systems*, 5(1):47–76, February 1987.

[12] G. Carle. Reliable group communication in atm networks. In *Proceedings of the Twelve Annual Conference on European Fibre Optic Communications and Networks EFOC&N'94*, June 21-24 1994.

[13] G. Carle and S. Dresler. Adaptable error control for efficient provision of reliable services in atm networks, December 1995. First Workshop on ATM Traffic Management WATM'95, IFIP, WG.6.2 Broadband Communication, Paris, 6.-8. December 1995.

[14] I. Cidon, T. Hsiao, A. Khamisy, A. Parekh, R. Rom, and M. Sidi. OPENET: An Open and Efficient Control Platform for ATM Networks. December 1995.

[15] Cornell University. *The CU-SeeMe Home Page, URL: http://cu-seeme.cornell.edu/.*

[16] D. Waitzman, C. Partridge, and S. Deering. *Distance Vector Multicast Routing Protocol.* IETF, November 1988.

[17] Day, J.D. and Zimmermann, H. The OSI reference model. In *Proceedings of the IEEE*, volume 71, pages 1334–1340, December 1983.

[18] S. Deering. *Host Extensions for IP Multicasting, RFC 1112.* Stanford University, August 1989.

[19] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. submitted for publication, 1995.

[20] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[21] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. To appear in IEEE/ACM Transactions on Networking.
An earlier version of this paper appeared in ACM SIGCOMM 95, August 1995, pp. 342-356., November 1995.

[22] S. Paul, K. K. Sabnani, and D. M. Kristol. Multicast transport protocols for high speed networks. AT&T Bell Laboratories, 1993.

[23] Romanow A. and Floyd S. Dynamics of TCP Traffic over ATM Networks. *IEEE JSAC*, 13(4):633–641, May 1995.

[24] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Department of Computer Science, August 1994.

[25] W. Vogels. World Wide Failures. Dept. of Computer Science, Cornell University, 1996.

[26] A. Yair, D. Breitgand, D. Dolev, and G. Chockler. Group communication as an infrastructure for distributed system management. In *Proceedings of the Third International Workshop on Services in Distributed and Networked Environments (SDNE'96)*, June 1996.

```
procedure LMS_RESOLVE_REPLY_HANDLER(m):

    $open\_request$ = FIND_OPEN_REQUEST_BY_ID($m.message\_id$);
    if ($open\_request$ == NULL)
        /* This might happen if the GMS was suspected to be failed and
          * the queue was reset because of that.
          */
        discard ($m$);
    else {
        /* **note** that there would be only **one** relevant reply:
          * the GMS would collect all the replies from the
          * neighbors that have information on the group in theirs
          * domains, and only after the full information was
          * obtained, it would forward the aggregated reply to the
          * LMS. Thus only one full reply should be received.
          * If the enqueued request is of the type RESOLVE_JOIN,
          * then it is a reconnection to the GMS that initiated this
          * resolve, and the reply will be forwarded as a EP_JOIN message.
          */
        if ($open\_request.type$ == RESOLVE_JOIN)
            DELIVERY($open\_request.waiting\_set$, EP_JOIN, $m.group$, $m.reply$);
        else {
            /* add the local membership to the total reply */
            $m.reply = m.reply$
                $\bigcup$ LOCAL_END_POINTS_OF_GROUP($m.group.group\_name$, $m.group.scope$, NULL);
            DELIVERY($open\_request.waiting\_set$, **absolute**, $m.group$, $m.reply$);
        }
        DEQUEUE_RESOLVE_REQUEST($open\_request.id$) ;
    }
```

Figure 13: The LMS Incoming Resolve Reply Message Handler.

```
procedure HANDLE_REGISTRATION_UPDATE_MSG(m):

    switch (m.type) {
        case ep_join or ep_leave:
            local-set =
                $\bigcup_{g \in m.groups}$ LOCAL_END_POINTS_OF_GROUP($g.group\_name, g.scope$, SET_ONLINE);

            DELIVERY(local-set, m.type, g, m.addresses_list);
        case hd_failure:
            /* m holds the server's address and its level , and list
             * of groups which had members in this domain.
             */
            $list\_of\_groups = m.groups$;
            $address\_prefix$ = MAKE_PREFIX($m.s\_address, m.level$);
            local-set= {};
            for each $group \in list\_of\_groups$ do
                local-set=local-set
                $\bigcup$ LOCAL_END_POINTS_OF_GROUP($group.group\_name, group.scope$, SET_ONLINE);

            DELIVERY(local-set, $m.failure\_type$, $address\_prefix$);
            /* When the $m.failure\_type$ is FILTER_OUT, the end-points have to remove all the
             * members with the same domain prefix from their membership. If the $m.failure\_type$
             * is FILTER_IN, then the end-points have to KEEP in theirs membership
             * ONLY those members with the SAME domain prefix as the $address\_prefix$
             */
            }
    } /* switch m.type */
```

Figure 14: The LMS Registration Message Handler.

```
procedure UPDATE_END_POINT_ONLINE_STATE($ep$, $group\_name$, $scope$, $online\_state$):

    if (¬MEMBER($ep, group\_name, scope$)
        return;
    $memb\_list$ = LOCAL_END_POINTS_OF_GROUP($group\_name, scope,$ NULL);
    if ($\exists$ $end\text{-}point$ $\in memb\_list$ with ON-LINE bit set to SET_ONLINE)
        $prev\_state$ = SET_ONLINE;
    else $prev\_state$ = RESET_ONLINE;
    UPDATE_GROUPS_INFO(SET_ONLINE, $group\_name, scope, ep, online\_state$);
    if ($\exists$ $end\text{-}point$ $\in memb\_list$ with ON-LINE bit set to SET_ONLINE)
        $current\_state$ = SET_ONLINE;
    else $current\_state$ = RESET_ONLINE;
    if ($prev\_state \neq current\_state$) {
        $online\_state\_msg$ = NEW(update_online_state);
        $online\_state\_msg.group.group\_name = group\_name$;
        $online\_state\_msg.group.scope = scope$;
        if ($current\_state$ == SET_ONLINE)
            $online\_state\_msg.state$ = ALL;
        else $online\_state\_msg.state$ = RESOLVE;
        if ($gms\_status$ == ON)
            SEND($\{my\_gms\}$, $online\_state\_msg$);
    }
```

Figure 15: The LMS Online State Update Handler.

```
procedure LMS_EP_FAILURE_HANDLER(ep):

    $list\_of\_groups$ = GROUPS_OF_LOCAL_END_POINT($ep$);
    $m$ = NEW(registration_update);
    $m.type$ = EP_LEAVE;
    $m.addresses\_list = \{ep\}$;
    $m.groups = list\_of\_groups$;
    if ($gms\_status$ == ON)
        SEND($\{my\_gms\}$ , $m$);
    /* Update open resolve requests queue due to the $ep$ failure */
    for each $request \in Open\_resolve\_requests\_queue$ do {
        if ($ep \in request.waiting\_set$) {
            $request.waiting\_set = request.waiting\_set \setminus \{ep\}$;
            if ($request.waiting\_set$ == {})
                DEQUEUE_RESOLVE_REQUEST($request.id$);
        }
    }
    for each $group \in list\_of\_groups$ do {
        UPDATE_END_POINT_ONLINE_STATE($ep$, $group.group\_name$, $group.scope$, RESET_ONLINE);
        UPDATE_GROUPS_INFO(DELETE, $group.group\_name, group.scope, ep,$ NULL);
        DELIVERY(LOCAL_END_POINTS_OF_GROUP($group.group\_name$, $group.scope$, SET_ONLINE),
                EP_LEAVE, $group$, $ep$);
    }
```

Figure 16: The LMS End-Point Failure Handler.

```
procedure LMS_HANDLE_GMS_FAILURE():

    $gms\_status = \text{OFF};$
    $local\text{-}set =$
        $\bigcup_{g \in \text{GROUPS\_OF\_LMS()}} \text{LOCAL\_END\_POINTS\_OF\_GROUP}(g.group\_name, g.scope, \text{SET\_ONLINE});$

    $\text{DELIVERY}(local\text{-}set,\ \text{FILTER\_IN},\ lms\_address);$
    $m = \text{NEW}(\textbf{resolve\_reply});$
    for each $req \in Open\_resolve\_requests\_queue$ do {
        $local\text{-}end\text{-}points = req.waiting\_set;$
        $m.group = req.group;$
        $m.reply = \text{LOCAL\_END\_POINTS\_OF\_GROUP}(m.group.group\_name, m.group.scope, \text{NULL});$
        $\text{DELIVERY}(local\text{-}end\text{-}points,\ \textbf{absolute},\ m.group,\ m.reply);$
        $\text{DEQUEUE\_RESOLVE\_REQUEST}(req.id);$
    }
```

Figure 17: The LMS Handler of The GMS Failure.

```
procedure LMS_RECONNECT_GMS():

    gms_status = ON;
    foreach group ∈ GROUPS_OF_LMS() do {
        memb_list = LOCAL_END_POINTS_OF_GROUP(group.group_name, group.scope, NULL);
        if (∃ end-point ∈ memb_list with ON-LINE bit set to SET_ONLINE)
        /* since there is at least one end-point on the LMS's
          * host that requires the membership notifications,
          * then the GMS must be notified about this - using a
          * global online flag.
          */
            online_flag = ALL;
        else online_flag = RESOLVE;
        join_msg = NEW(registration_update);
        join_msg.type = EP_JOIN;
        join_msg.addresses_list = memb_list;
        join_msg.groups = {group};
        SEND({my_gms} , join_msg);
        online_state_msg = NEW(update_online_state);
        online_state_msg.group = group;
        online_state_msg.state = online_flag;
        SEND({my_gms} , online_state_msg);
        put all members ∈ memb_list with ON-LINE bit set to SET_ONLINE,
            into waiting_end_points_set;
        if (waiting_end_points_set ≠ {}) {
            resolve_msg = NEW(resolve_request);
            resolve_msg.group = group;
            ENQUEUE_RESOLVE_REQUEST(resolve_msg.message_id, group, waiting_end_points_set,
                                    {my_gms}, RESOLVE_JOIN);
            SEND({my_gms} , resolve_msg);
        }
    }
```

Figure 18: The LMS Reconnection to the GMS Handler.

```
procedure GMS-MAIN-LOOP():

    Loop forever {
        switch (event) {
            case fault suspicion of neighbor $n$ :
                HANDLE_FAULT_SUSPICION($n$);
            case receive resolve_request message $m$ for group $g$ from neighbor $n$:
                HANDLE_RESOLVE_REQUEST($m, n$);
            case receive resolve_reply $r$ on resolve request from neighbor $n$:
                HANDLE_RESOLVE_REPLY($r, n$);
            case receive registration_update message $m$ from neighbor $n$:
                switch (m.type) {
                    case EP_JOIN:
                        HANDLE_EP_JOIN($m, n$);
                    case EP_LEAVE:
                        HANDLE_EP_LEAVE($m, n$);
                    case HD_FAILURE: /* host or domain failure */
                        HANDLE_HD_FAILURE($m, n$);
                }
            case receive update_online_state message $m$ from neighbor $n$:
                HANDLE_ONLINE_STATE_MESSAGE($m, n$);
            case detect reincarnation of neighbour $n$:
                HANDLE_SERVER_REINCARNATION($n$);
        }
    }
```

Figure 19: The GMS Main Event Loop.

```
procedure HANDLE_FAULT_SUSPICION(n):

    if (NEIGHBOR_STATE($n$) == DISCONNECTED) return;
    $affected\_groups$ = GROUPS_OF_NEIGHBOR($n$);
    SET_NEIGHBOR_STATE($n$, DISCONNECTED);
    if ($n \in$ CHILDREN($my\_level$)) {
        $tmp$ = NEIGHBORS_SET($my\_level$);
        if ($\nexists g \in affected\_groups$ s.t SCOPE_TO_LEVEL($g.scope$) $\geq my\_level + 1$)
            $tmp = tmp \setminus \{$PARENT($my\_level$)$\}$;
        $level = my\_level - 1$;
        $failure\_type$ = FILTER_OUT;
    }
    else {
        $tmp$ = CHILDREN($my\_level$);
        if ($n$ == PARENT($my\_level$)) {
            $level = my\_level + 1$;
            $failure\_type$ = FILTER_IN;
        }
        else {
            $level = my\_level$;
            $failure\_type$ = FILTER_OUT;
        }
    }
    $\mathcal{TG} = \{n' | n' \in tmp \bigwedge($NEIGHBOR_STATE($n'$) $\neq$ DISCONNECTED)
                    $\bigwedge($GROUPS_OF_NEIGHBOR($n'$) $\bigcap affected\_groups \neq \{\})\}$;
    $m$ = NEW(**registration_update**);
    $m.type$ = HD_FAILURE;
    $m.s\_address = n$;
    $m.level = level$;
    $m.failure\_type = failure\_type$;
    $m.groups = affected\_groups$;
    SEND($\mathcal{TG}$ , $m$);
    for each $r \in Open\_resolve\_requests\_queue$ do
        if ($n == r.from$) DEQUEUE_RESOLVE_REQUEST($r.id$);
        else if ($n \in r.waiting\_set$) {
            $r.waiting\_set = r.waiting\_set \setminus \{n\}$;
            if ($r.waiting\_set == \{\}$) {
                /* all operational entities have replied */
                $m'$ = NEW(**resolve_reply**);
                $m'.message\_id = r.id$;
                $m'.group = r.group$;
                $m'.addresses\_list = r.reply$;
                SEND($r.from$, $m'$);
                DEQUEUE_RESOLVE_REQUEST($r.id$);
            }
        }
        else {
            $failed\_domain$ = MAKE_PREFIX($n, level$);
            if ($failure\_type$ == FILTER_OUT)
                $r.reply = \{ep | ep \in r.reply \bigwedge \neg$IN_ADDRESS_SCOPE($ep, failed\_domain$)$\}$;
            else
                $r.reply = \{ep | ep \in r.reply \bigwedge$ IN_ADDRESS_SCOPE($ep, failed\_domain$)$\}$;
        }
```

Figure 20: The GMS Fault Suspicion Handler.

```
procedure HANDLE_SERVER_REINCARNATION(n):

    SET_NEIGHBOR_STATE($n$, CONNECTED);
    for each $group \in Groups$ do
        $Groups($INDEX_BY_GROUP$(group), n) =$ ALL;
    if ($n \in$ CHILDREN$(my\_level)$)
    /* if it's a child, then the re-merge process would be
     * triggered by the child himself, nothing to do here...
     */
      return;
    if ($n ==$ PARENT$(my\_level)$)
        $\mathcal{MG} = \{g | g \in Groups \bigwedge$ SCOPE_TO_LEVEL$(g.scope) > my\_level\}$;
    else /* its a sibling */
        /* all the groups in $Groups$ matrix has a scope at least of $my\_level$ */
        $\mathcal{MG} = \{g | g \in Groups\}$;
    for each $group \in \mathcal{MG}$ do {
        $i =$ INDEX_BY_GROUP$(group)$;
        $\mathcal{TG} = \{c | c \in$ CHILDREN$(my\_level) \bigwedge$ NEIGHBOR_STATE$(c) \neq$ DISCONNECTED
                                $\bigwedge (Groups(i, c) ==$ ALL $\bigvee Groups(i, c) ==$ RESOLVE$)\}$;
        if ($\mathcal{TG} \neq \{\}$) {
            $m =$ NEW(**resolve**);
            $m.group = group$;
            /* the reply to $m$ will be forwarded to $n$, as a JOIN message */
            ENQUEUE_RESOLVE_REQUEST$(m.message\_id, group, n, \mathcal{TG},$ RESOLVE_JOIN$)$;
            SEND$(\mathcal{TG}, m)$;
        }
    }
```

Figure 21: The Server Reincarnation Handler.

```
procedure HANDLE_RESOLVE_REQUEST(m,n):

    $g = m.group$;
    $group\_index$ = INDEX_BY_GROUP($g$);
    if ($group\_index == -1$)
        $send\_to\_all = True$;
    if ($n == parent(my\_level)$ or $n \in siblings(my\_level)$)
        $\mathcal{TG} = \{n'|n' \in$ CHILDREN$(my\_level) \bigwedge$ NEIGHBOR_STATE$(n') \neq$ DISCONNECTED
                                $\bigwedge(send\_to\_all \bigvee Groups(group\_index, n') \neq$ NONE$)\}$;
    else {/* $n \in children(my\_level)$ */
        if (SCOPE_TO_LEVEL$(g.scope) \geq (my\_level + 1)$)
            /* include the parent in target set */
            $\mathcal{TG} = \{n'|n' \in$ (SIBLINGS$(my\_level) \bigcup \{$PARENT$(my\_level)$)
                                $\bigwedge$ NEIGHBOR_STATE$(n') \neq$ DISCONNECTED
                                $\bigwedge(send\_to\_all \bigvee Groups(group\_index, n') \neq$ NONE$)\}$;
        else
            $\mathcal{TG} = \{n'|n' \in$ SIBLINGS$(my\_level) \bigwedge$ NEIGHBOR_STATE$(n') \neq$ DISCONNECTED
                                $\bigwedge(send\_to\_all \bigvee Groups(group\_index, n') \neq$ NONE$)\}$;
        if ($my\_level == 0$)
            $\mathcal{TG} = \mathcal{TG} \bigcup \{n'|n' \in$ CHILDREN$(my\_level) \bigwedge$ NEIGHBOR_STATE$(n') \neq$ DISCONNECTED
                                $\bigwedge(send\_to\_all \bigvee Groups(group\_index, n') \neq$ NONE$)\} \setminus \{n\}$;
    }
    if ($\mathcal{TG} == \{\}$) {
        $r$ = NEW(**resolve_reply**);
        $r.message\_id = m.message\_id$;
        $r.group = m.group$;
        $r.addresses\_list = \{\}$;
        SEND({m.from}, $r$);
    }
    else {
        ENQUEUE_RESOLVE_REQUEST$(m.message\_id, g, \{m.from\}, \mathcal{TG}$, RESOLVE);
        SEND$(\mathcal{TG}, m)$;
    }
```

Figure 22: The GMS Resolve Request Handler.

```
procedure HANDLE_RESOLVE_REPLY(r,n):

    /* find the open resolve request whose id matches the reply's id */
    $req\_info$ = FIND_OPEN_REQUEST_BY_ID($r.message\_id$);
    if ($req\_info$ == NULL)
    /* A reply for a request that was already handled is received from a neighbor.
     * This can happen if the neighbor was suspected to be disconnected.
     */
        return;
    $g = req\_info.group$;
    if (($r.end\_points\_list$ == {}) and (INDEX_BY_GROUP($g$) $\neq -1$))
        /* update groups matrix:
         */
        $Groups$(INDEX_BY_GROUP($g$), $n$) = NONE;
    /* add the current resolve reply to the total reply */
    $req\_info.reply = req\_info.reply \bigcup r.end\_points\_list$;
    /* remove the reply source from the waiting list of
     * the open resolve request
     */
    $req\_info.waiting\_set = req\_info.waiting\_set \setminus \{n\}$;
    if ($req\_info.waiting\_set$ == {}) {
        if ($req\_info.type$ == RESOLVE_JOIN) {
            /* convert the resolve reply into a JOIN message */
            $join\_msg$ = NEW(registration_update);
            $join\_msg.type$ = EP_JOIN;
            $join\_msg.addresses\_list = req\_info.reply$;
            $join\_msg.groups = \{g\}$;

            $online\_state\_msg$ = NEW(update_online_state);
            $online\_state\_msg.group = \{g\}$;
            if ($\exists c \in$ CHILDREN($my\_level$) $s.t$ $Groups$(INDEX_BY_GROUP($g$), $c$) == ALL)
                $online\_state\_msg.state$ = ALL;
            else
                $online\_state\_msg.state$ = RESOLVE;
            if ($req\_info.from$ == PARENT($my\_level$))
                /* if the destination for this resolve/join is the parent
                 * then mark the join message as a join_and_resolve message,
                 * else, mark it as a usuall join message.
                 */
                $join\_msg.join\_and\_resolve$ = TRUE;
            else
                $join\_msg.join\_and\_resolve$ = FALSE;
            SEND($req\_info.from$, $join\_msg$);
            SEND($req\_info.from$, $online\_state\_msg$);
        }
        else {
            $r$ = NEW(resolve_reply);
            $r.message\_id = req\_info.id$;
            $r.group = req\_info.group$;
            $r.addresses\_list = req\_info.reply$;
            SEND($req\_info.from$, $r$);
        }
        DEQUEUE_RESOLVE_REQUEST($req\_info.id$);
    }
```

Figure 23: The GMS Resolve Reply Handler.

```
procedure HANDLE_ONLINE_STATE_MESSAGE(m,n):

    $g = m.group$;
    $group\_index = \textsc{index\_by\_group}(g)$;
    if ($group\_index == -1$)
        return;
    $prev\_outer\_state = \textsc{get\_online\_state}(g, \{\textsc{parent}(my\_level)\} \bigcup \textsc{siblings}(my\_level))$;
    $prev\_inner\_state = \textsc{get\_online\_state}(g, \textsc{children}(my\_level))$;
    $Groups(\textsc{index\_by\_group}(g), n) = m.online\_state$;
    $current\_outer\_state = \textsc{get\_online\_state}(g, \{\textsc{parent}(my\_level)\} \bigcup \textsc{siblings}(my\_level))$;
    $current\_inner\_state = \textsc{get\_online\_state}(g, \textsc{children}(my\_level))$;
    if ($prev\_outer\_state \neq current\_outer\_state$ or $prev\_inner\_state \neq current\_inner\_state$) {
        if ($n == \textsc{parent}(my\_level)$ or $n \in \textsc{siblings}(my\_level)$)
            if ($my\_level == 0$)
                /* No need to update the LMS's about the change in the GMS state./*
                return;
            else
               $\mathcal{TG} = \{n' | n' \in \textsc{children}(my\_level) \bigwedge \textsc{neighbor\_state}(n') \neq \textsc{disconnected}$
                                                     $\bigwedge Groups(group\_index, n') \neq \textsc{none}\}$;
        else
            $\mathcal{TG} = \{n' | n' \in (\{\textsc{parent}(my\_level)\} \bigcup \textsc{siblings}(my\_level))$
                              $\bigwedge \textsc{neighbor\_state}(n') \neq \textsc{disconnected}$
                              $\bigwedge Groups(group\_index, n') \neq \textsc{none}\}$;
        $\textsc{send}(\mathcal{TG}, m)$;
    }
```

Figure 24: The GMS Online State Messages Handler.

```
procedure GROUPS_OF_NEIGHBOR(n):

    $list\_of\_groups = \{g | (\textsc{index\_by\_group}(g) \neq -1)$
                         $\bigwedge (Groups(\textsc{index\_by\_group}(g), n) \neq \textsc{none})\}$;
    return ($list\_of\_groups$);
```

Figure 25: Routine For Determining The Groups A Neighbor Interests In.

```
procedure HANDLE_EP_JOIN(m ,n):

    $g = m.groups$; /* in this case the groups list contains only a single name! */
    $req\_list$ = FIND_OPEN_REQUESTS_FOR_GROUP($g.group\_name, g.scope$);
    for each $r \in req\_list$ do
        if (($r \neq$ NULL) and ($n \in r.full\_waiting\_set$))
            $r.reply = r.reply \bigcup m.addresses\_list$;
    if (INDEX_BY_GROUP($g$) $== -1$)
        INSERT_NEW_GROUP($g$);
    if ($n ==$ PARENT($my\_level$) or $n \in$ SIBLINGS($my\_level$))
        $tmp\_set$ = CHILDREN($my\_level$);
    else { /* $n$ is a child */
        $tmp\_set$ = SIBLINGS($my\_level$);
        if (SCOPE_TO_LEVEL($g.scope$) $\geq (my\_level + 1)$)
            $tmp\_set = tmp\_set \bigcup$ PARENT($my\_level$);
        if ($my\_level == 0$)
            /* $n$ is an LMS, so add all the LMSs. */
            $tmp\_set = tmp\_set \bigcup$ CHILDREN($my\_level$) $\setminus \{n\}$;
    }
    $\mathcal{TG} = \{n' | n' \in tmp\_set \bigwedge$ NEIGHBOR_STATE($n'$) $\neq$ DISCONNECTED
                          $\bigwedge Groups(group\_index, n') ==$ ALL$\}$;
    if ($m.join\_and\_resolve ==$ TRUE) {
        if ($n \in$ SIBLINGS($my\_level$)) {
            /* No need to forward it as a 'join and resolve', since it will be
              * forwarded only to the children.
              */
            $m.join\_and\_resolve$ = FALSE;
            SEND($\mathcal{TG}$, $m$);
            $\mathcal{TG}' = \{n' | n' \in tmp\_set \bigwedge$ NEIGHBOR_STATE($n'$) $\neq$ DISCONNECTED
                    $\bigwedge (Groups(group\_index, n') ==$ RESOLVE $\bigvee Groups(group\_index, n') ==$ ALL$)\}$;
            $m'$ = NEW(**resolve_request**);
            $m'.group = g$;
            /* the reply to $m'$ will be forwarded to $n$, as a JOIN message */
            ENQUEUE_RESOLVE_REQUEST($m'.message\_id, g, n, \mathcal{TG}'$, RESOLVE_JOIN);
            SEND($\mathcal{TG}'$, $m'$);
        }
        else SEND($\mathcal{TG}$, $m$);
    }
    else SEND($\mathcal{TG}$, $m$);
```

Figure 26: The GMS End-Point Join Message Handler.

```
procedure HANDLE_EP_LEAVE(m ,n):

    $g = m.groups$; /* in this case the groups list contains only a single name! */
    $req\_list$ = FIND_OPEN_REQUESTS_FOR_GROUP($g.group\_name, g.scope$);
    for each $r \in req\_list$ do
        if (($r \neq$ NULL) and ($n \in r.full\_waiting\_set$))
            $r.reply = r.reply \setminus m.addresses\_list$;
    if ($n ==$ PARENT($my\_level$) or $n \in$ SIBLINGS($my\_level$))
        $tmp\_set$ = CHILDREN($my\_level$);
    else { /* $n$ is a child */
        $tmp\_set$ = SIBLINGS($my\_level$);
        if (SCOPE_TO_LEVEL($g.scope$) $\geq$ ($my\_level + 1$))
            $tmp\_set = tmp\_set \bigcup$ PARENT($my\_level$);
        if ($my\_level == 0$)
            /* $n$ is an LMS, so add all the LMSs. */
            $tmp\_set = tmp\_set \bigcup$ CHILDREN($my\_level$) $\setminus \{n\}$;
    }
    $\mathcal{TG} = \{n' | n' \in tmp\_set \bigwedge$ NEIGHBOR_STATE($n'$) $\neq$ DISCONNECTED
                          $\bigwedge Groups(group\_index, n') ==$ ALL$\}$;
    SEND($\mathcal{TG}$, $m$);
```

Figure 27: The GMS End-Point Leave Message Handler.

```
procedure HANDLE_HD_FAILURE(m ,n):

    if (($m.s\_address \in$ NEIGHBORS($my\_level$)) and (NEIGHBOR_STATE($m.s\_address$) == DISCONNECTED))
        return;
    /* $m$ holds the failed server (host/domain) address */
    $address = m.s\_address$;
    if ($address \in$ NEIGHBORS($my\_level$)) {
        $m.groups = m.groups \bigcup$ GROUPS_OF_NEIGHBOR($address$);
        SET_NEIGHBOR_STATE($m.s\_address$, DISCONNECTED);
    }
    $list\_of\_groups = m.groups$;
    for $g \in list\_of\_groups$ do {
        $req\_list$ = FIND_OPEN_REQUESTS_FOR_GROUP($g.group\_name, g.scope$);
        for each $r \in req\_list$ do
            if (($r \neq$ NULL) and ($n \in r.full\_waiting\_set$)) {
                $failed\_domain$ = MAKE_PREFIX($m.s\_address, m.level$);
                /* message is received from a child so $m.failure\_type$ == FILTER_OUT) */
                $r.reply = \{ep | ep \in r.reply \bigwedge \neg$IN_ADDRESS_SCOPE$(ep, failed\_domain)\}$;
            } /* if */
    } /* for */
    if ($n ==$ PARENT($my\_level$) or $n \in$ SIBLINGS($my\_level$))
        $tmp\_set$ = CHILDREN($my\_level$);
    else { /* $n$ is a child */
        $tmp\_set$ = SIBLINGS($my\_level$);
        if ($\exists g \in list\_of\_groups$ s.t SCOPE_TO_LEVEL($g.scope$) $\geq (my\_level + 1)$)
            $tmp\_set = tmp\_set \bigcup$ PARENT($my\_level$);
    }
    $\mathcal{TG} = \{n' | n' \in tmp\_set \bigwedge$ NEIGHBOR_STATE$(n') \neq$ DISCONNECTED
                    $\bigwedge(($GROUPS_OF_NEIGHBOR$(n') \bigcap list\_of\_groups) \neq \{\})\}$;
    SEND($\mathcal{TG}$, $m$);
```

Figure 28: The GMS Host/Domain Failure Message Handler.