Partitionable Light-Weight Groups*

Luís Rodrigues
Faculdade Ciências
Universidade de Lisboa
Campo Grande, Lisboa, Portugal
ler@di.fc.ul.pt

Katherine Guo
Lucent Bell Laboratories
101 Crawfords Corner Road Rm 4G-506
Holmdel, NJ 07733, USA
kguo@bell-labs.com

January 17, 2000

Abstract

Group communication, providing virtual synchrony semantics, is a powerful paradigm for building distributed applications. For applications that require a large number of groups, significant performance gains can be attained if these groups share the resources required to provide virtual synchrony. A service that maps multiple user groups onto a small number of instances of a virtually synchronous implementation is called a Light-Weight Group Service.

This paper describes the design of a light-weight group service able to operate in partitionable networks. Partitions pose challenges to the design of this service, in particular because inconsistent mapping decisions can be made when the system is partitioned. The paper focuses on the design of reconciliation mechanisms needed when a partition is healed.

1 Introduction

For developing distributed applications, virtually synchronous group communication [4, 15] is a powerful paradigm. It allows processes to be organized in *groups* within which messages are exchanged to achieve a common goal. Virtual synchrony ensures that all processes in the group receive consistent information about the group membership in the form of *views*. The

*This work was partially supported by Praxis/ C/ EEI/ 12202/ 1998, TOPCOM and by Fundação Oriente. Sections of this report will be published in Proceedings of the Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, Taipe, Taiwan, April, 2000.

membership of a group may change over time because new processes may join the group and old processes may fail or voluntarily leave the group. Virtual synchrony also orders messages with view changes, and guarantees that all processes that install two consecutive views deliver the same set of messages between these views. The usefulness of virtually synchronous semantics have been demonstrated in several areas, such as Computer Supported Cooperative Work [13], databases [10] and fault-tolerant distributed object systems [5].

In order to provide virtual synchrony, failure detectors and protocols which provide agreement and ordering are needed. Naturally, these components consume some amount of system resources such as bandwidth and processing power, but the overall performance impact is usually small. Opportunities for optimization occur when several groups have a large percentage of common members, because these groups can share common services. Such opportunities arise in several real-world applications. The Swiss Exchange Trading System [11] uses a group based system for data access and dissemination. A different group is associated with a different data "subject" and the resulting system uses as many as 50 groups that may overlap. Another example is CCTL [13], a group communication based collaboration system that manages several groups on behalf of the same application.

One way to achieve this benefit is to map several user level groups onto a single virtually synchronous group. Since the groups share common resources, they are subject to lower overhead than stand-alone groups and are consequently called *Light-Weight Groups* (LWGs). In contrast, the underlying virtually synchronous group is called a Heavy-Weight Group (HWG). A *Light-Weight Group Service* maps LWGs onto HWGs.

In previous work, we have designed and implemented a transparent and dynamic light-weight group service in non-partitionable networks [14, 8]. In this paper, we propose extensions to address the issue of network partitions. Network partition poses several challenges to the design of a light-weight group service. In particular, it is impossible to ensure the consistency of mapping decisions made in distinct partitions. Thus, partitionable operation requires certain reconciliation mechanisms to be executed when a partition heals. In addition, our original design assumes the availability of a naming service where mappings are stored, therefore, the naming service must also be augmented to tolerate partitions.

The paper is organized as follows. Related work is surveyed in Section 2. For self-containment, the design of the transparent and dynamic light-weight group service in non-partitionable environments is summarized in Section 3. The extensions required to tolerate partitions are introduced in Section 4, then the auxiliary support services are described in Section 5 and the operation of the partitionable light-weight group service is presented in Section 6. Section 7 concludes the paper.

2 Related Work

The light-weight group service idea can be traced to several sources. In the Delta-4 group communication subsystem, several session level groups could be mapped statically onto a single virtually synchronous MAC level group [12]. The Isis system extended this principle, offering a light-weight group service that supports dynamic associations between user level

groups and core Isis groups [7]. However, the Isis LWGs require the specification of the target membership of a user group to make appropriate mapping decisions. Neither of these approaches is transparent, because they force the application designer to be aware of details that are related with the way the service is provided.

Some light-weight group approaches have all groups in the system share some resources, such as a failure detector or an underlying ordered channel [1]. Another approach, followed in CCTL [13] and Maestro [3], consists of building a "coordination" layer to manage a collection of different protocol stacks. Because these approaches implement a static form of a light-weight group service, they do not address the problem of minimizing interference among unrelated groups.

In previous work we have addressed the issue of building a transparent light-weight group service in non-partitionable networks [14, 8]. The work addressed the following problems raised by the transparency requirements: i) the need to perform mappings between LWGs and HWGs in an automatic and dynamic manner; ii) the design of mechanisms to allow these mappings to be changed in run time. Another contribution of our previous work was to study the *interference* effect in light-weight group systems. Interference occurs when LWGs with different membership, or even non-overlapping LWGs , are mapped on the same HWG and events in one group affect the performance of another group. Sources of interference are the use of a common multicast transport channel, executions of failure recovery protocols, and the need to filter information at the LWG layer. Thus, the task of mapping LWGs onto HWGs addresses two conflicting goals: increasing resource sharing and minimizing interference.

To our knowledge, the issue of supporting a light-weight group service in partitionable networks has not been addressed previously in the literature. This work discusses difficulties introduced by network partitions and proposes new protocols to overcome them.

3 Light-Weight Groups in Non-Partitionable Networks

Informally, virtual synchrony provides group membership information to each process in the form of *views*. The model guarantees that all processes that install two consecutive views deliver the same set of messages between the views. The main goal of the transparent and dynamic LWG service is to support resource sharing by mapping multiple LWGs that have overlapping membership onto a single HWG , while fully preserving the original HWG interface when these LWGs are accessed. For self-containment, this section briefly describes the components of a transparent and dynamic LWG service in non-partitionable networks [14, 8].

The task of LWG service is performed by managing a pool of HWGs and establishing associations between LWGs and these HWGs . Each time a new LWG is created, the service determines whether the LWG can be associated with one of the existing HWGs (if any), if not, a new HWG is created and added to the pool. Whatever decision is made, the new LWG will be associated with some HWG and may subsequently share that HWG with other LWGs . Since the design imposes no restriction on the way the membership of LWGs changes in time, mappings that were appropriate at one point may become inefficient as the system

Requests(Downcalls)	
Name	Parameters
`Join`	GroupId gid, Pid pid
`Leave`	GroupId gid, Pid pid
`Send`	GroupId gid, BitArray data
`StopOk`	GroupId gid
Upcalls	
Name	Parameters
`View`	GroupId gid, PidList view
`Data`	GroupId gid, Pid src, BitArray data
`Stop`	GroupId gid

Table 1: VS interface primitives

evolves. To compensate, the LWG service dynamically redefines mappings as membership changes. When this happens, we say that a LWG is *switched* from one HWG to another.

As a result, the LWG service has three main tasks: (i) preserve the virtually synchronous interface of the HWGs to LWG users; (ii) define the mapping and switching policies; and (iii) invoke a *switching protocol*, which changes the association between a LWG and a HWG at run time.

3.1 Protocols

In order to preserve the original virtually synchronous interface, the LWG layer has to execute protocols that support the standard group operations. A typical interface of a virtually synchronous layer contains the following primitives, as listed in Table 1: `Join`, allows a member to join a group; `Leave`, allows a member to leave a group; `Send`, sends a virtually synchronous multicast; `View`, installs a new view; `Data`, indicates the delivery of a multicast; `Stop`, indicates that the traffic must be stopped temporarily (usually, when a view change in the virtually synchronous layer is in process); and `StopOk`, confirms the `Stop` indication. `Stop` and `StopOk` may be hidden from the user at upper layers.

The LWG protocols use the services of the underlying HWG group to provide service to LWG users with minimal overhead. Some of these protocols, such as the message passing protocol, are straightforward: the LWG `Send` service simply encapsulates the LWG message in a new ⟨`DATA, lwg_id, data`⟩ message which is multicast on the HWG; on the receiving side, when such message arrives, the `lwg_id` part is examined and the `data` part forwarded to the specified LWG. Other protocols, namely joining (performed in response to `Join`), leaving (performed in response to `Leave`), and switching (triggered internally by reconfiguration policies) are more complex since they must preserve virtual synchrony with minimal interference with the operation of other LWGs. The core of these protocols is a *flush* procedure, that makes sure that all in-transit messages are delivered before a new view is installed. The details of the protocols can be found in [14].

Name	Parameters	Returns
ns.set	LwgId lwg, HwgId hwg	none
ns.read	LwgId lwg	HwgId hwg
ns.testset	LwgId lwg, HwgId hwg	HwgId hwg

Table 2: Naming service interface primitives

An important aspect of the implementation of a light-weight group service is the coordination among processes to establish the same mappings for the LWGs, that is, in the same partition, all members of a given LWG must share exactly the same HWG. The implementation of the light-weight group service requires mappings between LWGs and HWGs to be stored in a way that can be accessed by every process. In this paper we assume that mappings are stored in an external *Naming Service*. The naming service exports three primitives, as illustrated in Table 2: `ns.set`, which establishes a mapping between a LWG and a HWG; `ns.read`, which returns the current mapping for a given LWG; and `ns.testset`, which returns the current mapping for a given LWG or, if no such mapping exits, establishes a new mapping to the HWG specified.

When a new mapping is established, the naming service is informed of the new mapping so that further joins are directed to the appropriate HWG. A problem of using an external naming service to keep the mapping information is that it is difficult to guarantee that processes always read up-to-date information. To avoid expensive synchronization procedures, we allow processes to read outdated information. To compensate for this, all members of a HWG keep information about the new mappings of previously mapped LWGs. This information is used like a forward-pointer, to redirect a process that is using outdated mapping information.

Note that, for availability, the naming service may be replicated. A possible implementation would replicate the naming service at every process, making updates expensive but read operations purely local. The use of the naming service in a partitionable network is discussed in Section 5.2.

3.2 The policies of dynamic light-weight group service

A dynamic light-weight group service is interesting in systems where a process is not required to know its future group membership in advance, as the membership often depends on run-time parameters like number and location of users, load, occurrence of faults and so on. Systems such as Isis [4], Horus [16], and Ensemble [3, 9] all share this feature. Thus, the LWG service must be able to operate without this *a priori* information, using heuristics to find the most appropriate mappings between LWGs and HWGs. An optimal mapping for a given set of LWGs is one that balances the twin goals of increasing resource sharing and minimizing interference. We briefly summarize the mapping policies used in [8].

When a LWG group is created, a mapping needs to be established. In our work we use an

optimistic approach that assumes the membership of the new LWG will be similar to some other already existing LWG. The new LWG is mapped onto some existing HWG and if the choice is later proven to be inappropriate, the LWG will be switched onto a more appropriate HWG. Due to the lack of information about the future, some of the mappings done at group creation time will later reveal to be disadvantageous. Corrective measures are based on the ability to change the mappings between LWGs and HWGs at run-time.

In general terms, the adaptive measures follow a number of simple guidelines:

- *Sharing rule.* Since the ultimate goal is to promote resource sharing, LWGs with similar membership should be mapped onto the same HWG. Keeping the number of HWGs low produces other advantages. When the number of HWGs is low, the search space is small and the heuristics can be applied in more efficient ways.
- *Interference rule.* To minimize interference, a LWG should be mapped onto a HWG with a similar membership.
- *Shrink rule.* Due to system evolution, it is possible that a process will find itself a member of a HWG without having any LWG mapped on it. If this situation persists for some time, the process should leave the HWG. Ultimately, this strategy causes a HWG with no LWG mapped onto it to be deleted.

These guidelines are applied using exclusively local heuristics to avoid using a central server that would be both a bottleneck and a single point of failure. The share, interference, and shrink rules are executed at every process and are based on comparing the membership of all LWGs and HWGs that are known to that process. The details of these rules are presented in Figure 1.

Poorly chosen local heuristics lead to instability, preventing the system from converging to a stable mapping. To avoid this problem, we have implemented a number of preventive measures, as described below. For each LWG, only one process is responsible for changing its mapping. This is the coordinator of the group (usually its oldest member). This strategy prevents different processes from making incompatible mapping decisions. For a given configuration, the mapping decision is deterministic. For instance, if several HWGs match a mapping criterion, the total order of group identifiers is used to make the selection. Following this approach, different invocations of the heuristics on the same configuration will always achieve the same results. We have selected the parameters so that a significant change in the membership must occur before a new mapping will be defined. Specifically, in the prototype we have set $k_m = 4$, $k_c = 4$ in Figure 1. In this setting, for a LWG to be mapped on a HWG, the number of their common members must be greater than 75% of the size of the HWG, and the mapping remains stable until this number is reduced to 25%. To avoid a cascade of switch events when groups are being created or deleted, the heuristics are applied periodically with a relative large period (in the prototype we ran them once every minute). This also makes the overhead of executing the heuristics and running the switch protocol negligible.

Definitions: *(k_m and k_c are configuration parameters)*
minority: given groups $g_1 \subseteq g_2$, g_1 is a minority of g_2 iff sizeof(g_1) $\leq$ sizeof(g_2)/k_m.
closeness: given $g_1 \subseteq g_2$, g_1 and g_2 are close enough to each other iff sizeof(g_2) - sizeof(g_1) $\leq$ sizeof(g_2)/k_c.

Share rule *(for some configuration parameter k_m)*
Considering a LWGs *pair (lwg_1, lwg_2) with (hwg_1, hwg_2) as their underlying* HWG *pair,*
where sizeof(hwg_1) = $n_1 + k$,
sizeof(hwg_2) = $n_2 + k$ and sizeof($hwg_1 \cap hwg_2$) = k.
`if` $\neg$ *((*$hwg_1 \subseteq hwg_2 \wedge hwg_1$ *is a minority of* hwg_2*)* $\vee$ *(*$hwg_2 \subseteq hwg_1 \wedge hwg_2$ *is a minority of* hwg_1*))*
$\wedge$ *(*$k > \sqrt{2n_1n_2}$*)* `then`
collapse hwg_1 and hwg_2 into a single hwg;
`fi`

Interference rule
Considering a LWG *lwg_1 with hwg_1 as its underlying* HWG*.*
`if` *(lwg_1 is a minority of hwg_1)* `then`
`if` *(*$\exists hwg_x$ *with membership close enough to* lwg_1*)* `then`
switch lwg_1 to hwg_x;
`else`
create a hwg_{new} with membership identical to lwg_1;
switch lwg_1 to hwg_{new};
`fi`
`fi`

Shrink rule
`for` *(each* HWG *member h)* `do`
`if` *(*$\neg$*(*$\exists$ a* LWG *mapped onto h))*`then` *h leaves its* HWG*;* `fi`
`od`

Figure 1: The local algorithms

3.3 Performance of light-weight groups

We have conducted a large number of experiments to evaluate the performance of the transparent and dynamic light-weight group service. The experiments were done using an implementation of the LWG service in Horus [16], on a system of SUN Sparc10 workstations running SunOS 4.1.3, connected by a loaded 10M bps Ethernet (the low level protocol we used is UDP/IP with IP multicast extension). For simplicity, we illustrate the result from one of several configurations reported in [8].

The configuration in Figure 2 presents the performance of two sets of n user groups where each group within a set has identical membership of 4 processes, and the two sets have disjoint membership. When no LWG service is used, each user group is mapped onto one virtually synchronous group. When a static LWG service is used, each user group is mapped onto a LWG of size 4 and each LWG is mapped onto the HWG consists of all the

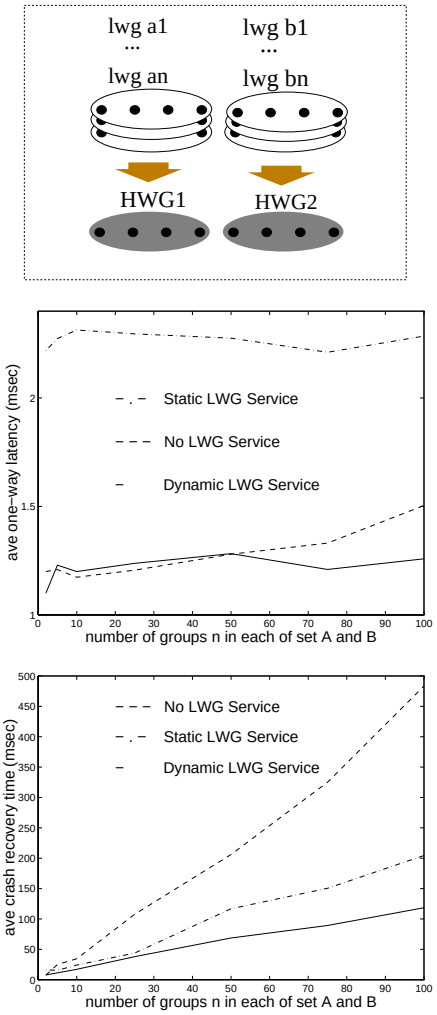

Figure 2: Performance of light-weight groups

8 processes. When a dynamic LWG service is used, LWGs a_1 to a_n are mapped onto the HWG with identical membership – HWG1, and LWGs b_1 through b_n are mapped onto HWG2. We have measured the latency and throughput of data transfer and the time to recover from the crash of a member. The results illustrate the effects of interference and resource sharing. In data transfer, a static LWG service is much worse than dynamic LWG service or even no LWG service at all due to problems of interference among unrelated groups. On the other hand, the advantage of having dynamic LWGs over having no LWG service are clear in the recovery time figure, which shows the benefits of resource sharing. Overall the dynamic LWG service provides the best performance because it balances resource sharing and interference reduction.

4 Partitionable Light-Weight Groups

Previous work on light-weight groups did not address the issue of network partitions [12, 7, 14, 8]. However, tolerance to network partitions is of practical importance when networks of large geographical scale are used. This type of networks may exhibit partitioned behavior due to crashes of routers or links. They can also exhibit what has been called *virtual partitions*, triggered by excessively loaded portions of the network, whose delays cause timeouts to expire and the connections to be marked as crashed. In asynchronous systems a virtual partition is indistinguishable from a network partition [2]. In practice, virtual partitions tend to disappear and "heal" faster than partitions caused by crashes and may result in higher system instability. To the best of our knowledge, this work is the first design of a light-weight group service that addresses network partitions.

Partitions raise several challenges to the design of a light-weight group service. In partitionable networks, group members must perform "peer-discovery" for partition healing. In other words, the group service must have the mechanisms to allow members to detect that they can communicate again (or for the first time) with other members that have been active in concurrent partitions. In our case, this task is complicated due to the fact that nodes in different partitions are unable to coordinate mapping decisions. This may lead to the establishment of different mappings for the same LWG in concurrent partitions. When these partitions merge, the light-weight group service is required to merge the concurrent views of the LWGs but, in order to do so, it also has to conciliate the mapping decisions. The following strategy is used to achieve these goals.

1. We rely on a partitionable HWG to offer us failure detection, "peer-discovery" at the HWG level, merging of HWG views, and virtually synchronous communication.

   This follows from the approach used in non-partitionable networks, that allows us to avoid "re-implementing" at the LWG layer complex protocols that are supported by the HWG layer. Therefore, the details of how partition healing is performed at the HWG become transparent to the LWG layer. This makes the LWG layer highly portable, since some of these mechanisms can be tailored to specific network classes without forcing any change in the light-weight group service.

2. We rely on an external naming service to provide up-to-date mappings between LWG views and HWG views. The naming service must be implemented using distributed cooperative servers in order to be able to provide service during network partitions. Special care is needed to re-conciliate the name servers' databases when a partition heals.

3. With the two previous mechanisms, once a HWG partition heals, the LWG can merge in the following steps:

Step 1: Using the re-conciliated state of the name server, members of concurrent LWG views mapped on different HWG become aware of one another.

Step 2: LWG views in Step 1 (if any) need to be re-mapped, in order to have all concurrent views of the same LWG mapped onto the same HWG. (Notice they can be mapped onto different views of the same HWG.)

Step 3: As a result of partition healing at the HWG, concurrent views of a LWG with a consistent mapping will share the same HWG view; they become aware of one another using a peer-discovery mechanism local to the HWG.

Step 4: Concurrent views of a LWG mapped on the same HWG view are merged into a single LWG view.

Section 5 introduces the two support mechanisms, namely the partitionable HWG service and the partitionable naming service. Section 6 describes each of the four steps in detail.

5 Support Services

5.1 Underlying partitionable group service

As previously discussed, the LWG layer is implemented on top of a HWG layer. We assume the HWG layer is able to operate in a partitionable environment. To avoid a long detour, we do not discuss how the definition of virtual synchrony can be extended to address partitions or describe protocols required to implement it. Interested readers can consult the rich bibliography on the subject (for instance, [1, 6, 2]). Instead, we assume that the HWG layer continues to deliver views in the presence of partitions, allowing groups to split into concurrent views when a partition occurs and these views to merge when the partition is healed.

Since in a partitionable system there could be multiple concurrent views of the same group, we identify each view using a *view identifier*, defined as a pair *(coordinator, view-sequence-number)*, where *view-sequence-number* is a local counter incremented by the coordinator of the view whenever a new view is installed. Each protocol message in the LWG service design is tagged with a view identifier when it is sent and is only delivered to members of that view. This allows to decouple the merge of the LWGs from the merge of the underlying HWG.

5.2 The naming service

The light-weight group layer needs to store the mappings between LWGs and HWGs. As mentioned in Section 3, we have opted to rely on an external naming service to store these mappings. If the naming service is implemented by a single server, then it becomes a bottleneck and a single point of failure, therefore limiting the scalability of the external naming service approach. However, the naming service simply abstracts many different ways to maintain a distributed database of LWG to HWG mappings, and should not affect the scalability of the LWG design.

In fact, in partitionable networks the naming service has to be implemented using a set of cooperative servers. The servers must be physically placed in strategic locations, to ensure that there is a high probability of having at least one server available at each partition (for instance, by placing a server in each autonomous system or on each local area network). Since coordination is not possible in the presence of partitions, strong replica consistency cannot be enforced on the name servers. In order to provide high degree of availability, our design of the naming service allows inconsistent mappings to coexist and provides support for reconciliation of mappings.

When name servers become reachable by other name servers after a network partition has been healed, a database reconciliation procedure needs to be performed. Mappings that are known in one view and not known in the other view are simply propagated. However, since conflicting mappings are unavoidable, the mapping information needs to be rich enough to avoid ambiguities. For partitionable operation, we augment the name service to include additional information about mappings. Instead of merely storing mappings between LWGs and HWGs , it stores mappings between specific LWG *views* and HWG *views*, that is, the naming service recognizes that concurrent group views can exist both at the LWG level and at the HWG level.

The example in Figure 3 illustrates this behavior. In partition p, the LWG view lwg_a is mapped onto HWG view hwg_1 and LWG view lwg_b is mapped onto HWG view hwg_2. In partition p', lwg'_a is mapped onto hwg'_2 and lwg'_b is mapped onto hwg'_1. When the partition heals, these concurrent views will coexist for some time, thus the name server will store both mappings as presented in Table 3.

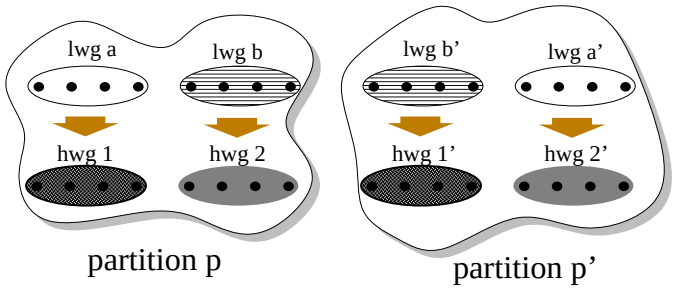

Figure 3: Inconsistent mappings

In response to the reconciliation, the naming service can detect inconsistent mappings. However, it is not the role of the naming service to conciliate these mappings although it can provide some help, by notifying the relevant LWGs , as it will be explained in the next section.

When two concurrent views are merged (either HWG views or LWG views), a new view-to-view mapping is defined, thus the naming service needs to be notified to update the database. Note that, in absence of further partitions, the system should converge to a scenario where a single merged view of a LWG is mapped onto exactly one HWG . A typical evolution from Figure 3 would be: the views of the HWGs would merge; the views of the LWGs would be migrated such that concurrent views are mapped onto the same HWG view; finally concur-

partition p	partition p'
LWG $_a$: $lwg_a \rightarrow hwg_1$ LWG $_b$: $lwg_b \rightarrow hwg_2$	LWG $_a$: $lwg'_a \rightarrow hwg'_2$ LWG $_b$: $lwg'_b \rightarrow hwg'_1$
merged naming service	
LWG $_a$: $lwg_a \rightarrow hwg_1$, $lwg'_a \rightarrow hwg'_2$ LWG $_b$: $lwg_b \rightarrow hwg_2$, $lwg'_b \rightarrow hwg'_1$	

Table 3: Merging the naming service database

rent views of the LWGs would be merged. This evolution is illustrated in Figure 4 and the associated content of the naming service is presented in Table 4.

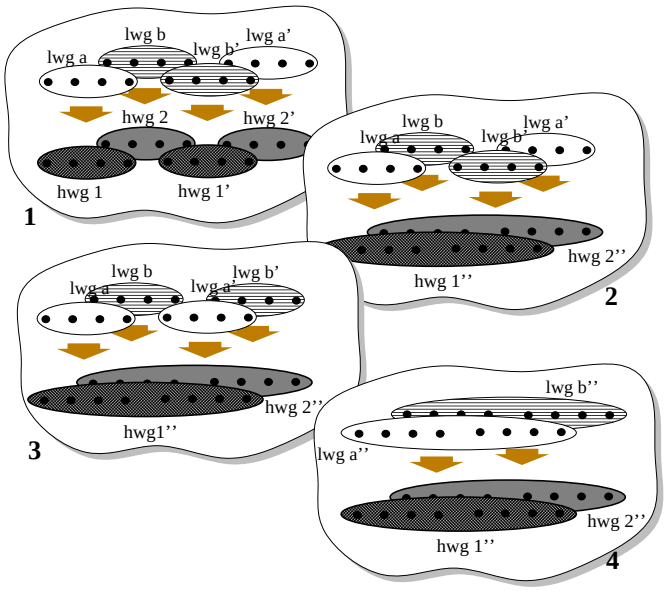

Figure 4: Partition merge. A typical evolution from Figure 3.

Note that, in order to discard obsolete information from the database, the naming service must be aware of the partial order of views. For instance, in Figure 4, when a mapping for lwg''_a is stored in the database, the naming service needs to be informed that view lwg''_a is the result of merging lwg_a and lwg'_a. Therefore, the mappings for these previous views are deleted from the database.

6 Operation of the Partitionable LWGs

Whenever a partition heals at the HWG level, LWG members go through the following four steps to merge the partition at the LWG level: *i)* discover other concurrent LWG views with different mappings (if any) through global "peer discovery"; *ii)* reconcile different mapping

1) merged naming service
LWG $_a$: $lwg_a \rightarrow hwg_1$, $lwg'_a \rightarrow hwg'_2$
LWG $_b$: $lwg_b \rightarrow hwg_2$, $lwg'_b \rightarrow hwg'_1$
2) merged HWGs
LWG $_a$: $lwg_a \rightarrow hwg''_1$, $lwg'_a \rightarrow hwg''_2$
LWG $_b$: $lwg_b \rightarrow hwg''_2$, $lwg'_b \rightarrow hwg''_1$
3) switched LWGs
LWG $_a$: $lwg_a \rightarrow hwg''_1$, $lwg'_a \rightarrow hwg''_1$
LWG $_b$: $lwg_b \rightarrow hwg''_2$, $lwg'_b \rightarrow hwg''_2$
4) merged LWGs
LWG $_a$: $lwg''_a \rightarrow hwg''_1$
LWG $_b$: $lwg''_b \rightarrow hwg''_2$

Table 4: Evolution of the naming service database

decisions in concurrent LWG views to map views of the same LWG onto the same HWG; *iii)* discover other concurrent LWG views mapped on the same HWG view; *iv)* merge concurrent LWG views of the same LWG into one.

6.1 Global peer discovery

Global peer discovery refers to identifying mappings of the same LWG to different HWGs in different partitions. Consider again the example in Figure 3. How can a member of lwg_a mapped on hwg_1 be aware of the existence of a concurrent view lwg'_a mapped on another HWG? One possible way is to require group members to periodically inquire one of the reachable name servers. Unfortunately, this could load the servers with unnecessary requests. Instead, we use the callback approach which requires the name servers to notify the affected groups whenever inconsistent mappings are detected during the name server reconciliation procedure. Again, in our example, after the reconciliation the name server will detect inconsistent mappings and inform both concurrent views with a MULTIPLE-MAPPINGS message. The message contains all the mappings stored for the LWG in the name server.

6.2 Reconciliating inconsistent mappings

When the MULTIPLE-MAPPINGS message is received at a member, the switching procedure is triggered. We recall that only the coordinator of each view is responsible for initiating the switching of the view. To ensure that coordinators of concurrent views make the same decision, we enforce inconsistent mappings to be conciliated by switching to the HWG with highest group identifier. In our example, if $gid(hwg_1) < gid(hwg_2)$, the mapping should be switched to HWG $_2$; thus the view lwg_a needs to be switched and the view lwg'_a should keep

the same mapping. As a result of the switch, two concurrent views of LWG $_a$, lwg_a and lwg'_a are mapped onto the same HWG view – hwg''_2.

6.3 Local peer discovery

Local peer discovery refers to identifying different views of a LWG mapped onto the same HWG. When two concurrent LWG views are mapped onto the same HWG group view, peer discovery becomes straightforward. Recall that messages sent to a given LWG view are multicast on the underlying HWG and received by all HWG members. When a LWG message arrives at the LWG layer at a given node, the layer discards the message if there is no local member of the LWG. Otherwise it checks which view the local member is in. If the member is in the same view in which the message is sent, the LWG layer delivers the message to the LWG member as usual. If not, the member must be in some other concurrent view, and the LWG layer detects the presence of multiple concurrent LWG views and triggers the procedure to merge LWG views.

6.4 Merging light-weight group views

The final part of the reconciliation procedure is for concurrent LWG views, mapped on the same HWG view to be merged.

The concurrent views of LWGs mapped onto the same HWG can be merged by forcing a flush of the HWG and using this synchronization point to merge all LWG views at once. The protocol is initiated by making the node that triggers the procedure multicast a MERGE-VIEWS message to the HWG. Upon receipt of MERGE-VIEWS, each member multicasts a MAPPED-VIEWS message to the HWG, containing a list of current mappings. After receiving the first MERGE-VIEWS message, the coordinator of the HWG flushes the HWG [1]. The flush of the HWG will, in turn, force the flush of all concurrent LWG views and ensure that, by the end of the flush procedure, all members have received the same set of MAPPED-VIEWS messages. When the coordinator of the HWG installs a new view, concurrent views of the LWG are merged into a single view in a decentralized and deterministic way (since all processes have the same information).

The pseudo-code for the merge-views algorithm is presented in Figure 5. Note that the algorithm merges all concurrent views of all LWGs mapped in the same HWG in a single flush operation. Resource sharing is promoted because a flush for each light-weight group is avoided. Interference is avoided because other LWGs, mapped onto unrelated HWGs are not affected. Note also that when the LWG views are merged, the naming service is updated to produce the behavior illustrated by Table 4.

[1] Notice every node can send a MERGE-VIEWS message, the coordinator of the HWG starts the flush after receiving the first MERGE-VIEWS message and ignores further MERGE-VIEWS messages until a new view is installed.

```
100  variables:
101       V_p: set of light-weight views process p belongs to.
102       AV_p(hwg): all views mapped on hwg and known by p

103  when ⟨DATA, lwg, view, data⟩ received at p do
104       if view ∈ V_p then
105            deliver data to local member;
106       elseif ∃v ∈ V_p : v is concurrent with view then
107            multicast ⟨MERGE-VIEWS⟩ on the HWG.

108  when ⟨MERGE-VIEWS⟩ received at p do
109       multicast ⟨ALL-VIEWS, V_p⟩ on the HWG.
110       if p is the coordinator of the HWG then
111            force the flush of the hwg;

112  when ⟨ALL-VIEWS, V_q⟩ received at p do
113       let AV_p(hwg) := AV_p(hwg) ∪ V_q;

114  when the hwg is flushed do
115       merge all concurrent views in AV_p(hwg);
116       update V_p;
117       update the naming service;
118       deliver views and re-start groups.
```

Figure 5: Merge views protocol

7 Conclusions

When several groups have the same or similar membership, resource sharing can improve performance. Light-weight groups allow resource sharing by mapping several user level groups onto a single virtually synchronous group. The implementation of LWGs in partitionable environments is of practical importance but raises several challenges, namely because inconsistent mapping decision can be made in concurrent partitions.

In this paper we have discussed the reconciliation procedures that need to be performed when partitions are healed. A weakly consistent naming service that employees a specialized reconciliation algorithm is used to maintain mapping decisions. The naming service is loosely synchronized with the mechanisms used to merge concurrent views of light-weight groups. The paper also shows the importance of preserving information about the causal order of views: in our case this information is used to garbage collect obsolete information in the naming service.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proc. 22nd Annual International Symposium on Fault-Tolerant*

Computing, pages 76–84, July 1992.

[2] Ö. Babaoğlu, A. Davoli, A. Montresor, and R. Segala. System support fpr partition-aware network applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 184–191, Amsterdam, The Netherlands, May 1998. IEEE.

[3] K. Birman, R. Friedman, and M. Hayden. The Maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical Report TR97-1619, Dept. of Computer Science, Cornell University, Feb. 1997.

[4] K. Birman and T. Joseph. Exploiting replication in distributed systems. In S. Mullender, editor, *Distributed Systems*, pages 319–366. ACM Press Frontier Series, 1989.

[5] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, H. Sander, D. Bakken, M. Berman, D. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, West Lafayette, Indiana, USA, Oct. 1998.

[6] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical report, The Hebrew University of Jerusalem, 1993.

[7] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-weight process groups in the ISIS system. *Distributed System Engineering*, (1):29–36, 1993.

[8] K. Guo and L. Rodrigues. Dynamic light-weight groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 33–42, Balitmore, Maryland, USA, May 1997. IEEE.

[9] M. Hayden. *The Ensemble System.* PhD thesis, Cornell University, Ithaca, NY, Jan. 1998. Also available as Dept. of Computer Science Tech. Rep. TR98-1662.

[10] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[11] R. Piantoni and C. Stancescu. Implementing the swiss exchange trading system. In *Digest of Papers, The 27th International Symposium on Fault-Tolerant Computing Systems*, pages 309–313, Seattle, WA, July 1997.

[12] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing.* ESPRIT Research Reports. Springer Verlag, Nov. 1991.

[13] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed collaboration systems. In *Proc. 17th International Conference on Distributed Computing Systems*, pages 43–50, Balitmore, MD, May 1997.

[14] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Veríssimo, and K. Birman. A transparent light-weight group service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 130–139, Niagara-on-the-Lake, Canada, Oct. 1996.

[15] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 534–543, Toulouse, France, June 1993.

[16] R. van Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.