

Dynamic Light-Weight Groups*

Katherine Guo

Department of Computer Science

Cornell University

kguo@cs.cornell.edu

Luís Rodrigues

Departamento de Informática

Faculdade Ciências, University of Lisboa

ler@di.fc.ul.pt

Abstract

The virtual synchrony model for group communication has proven to be a powerful paradigm for building distributed applications. In applications that use 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 user groups onto instances of a virtually synchronous implementation is called a Light-Weight Group Service.

This paper discusses the Light-Weight Group protocols in dynamic environments, where mappings cannot be defined a priori and may change over time. We show that it is possible to establish mappings that promote sharing and, at the same time, minimize interference. These mappings can be established in an automated manner, using heuristics applied locally at each node. Experiments using an implementation in the Horus system show that significant performance improvements can be achieved with this approach.

*Selected section of this report were published in the Proceedings of the 17th IEEE International Conference on Distributed Computing Systems, Baltimore, Maryland, USA, May, 1997. This work was partially supported by the CEC, through ESPRIT/ BRA Working Group 26 (GODC) and the ARPA/ONR grant N00014-92-J-1866.

1 Introduction

Virtually synchronous group communication [2, 3, 11] has proven to be a powerful paradigm for developing distributed applications. This paradigm 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 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.

To provide virtual synchrony, failure detectors and protocols that 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 of these services is usually small. Opportunities for optimization occur when several groups have a large percentage of common members, by allowing these groups to share common services. Such opportunities appear in several real-world applications. In the INFS system, a reliable network file system built on the Isis system, the replicas for a file change over time as users change the replication properties of the file or as access patterns to the file change. The large number of files amortized over a small set of file replication servers causes a significant sharing of common services. Another example is the Orbix+Isis product from Isis Distributed Systems, and Iona Technologies Ltd, where the object-oriented programming style creates many object groups over a smaller set of processes or machines.

Mapping several user level groups onto a single virtually synchronous group can achieve this type of optimization. Since these groups share common resources, they can be implemented more efficiently than standalone groups and are called *Light-Weight Groups* (LWGs). In contrast, the underlying virtually synchronous group is called a Heavy-Weight Group (HWG) in this context. A service that maps LWGs onto HWGs is usually called a *Light-Weight Group Service*.

However, a tradeoff exists between resource sharing and interference. The operation of some other unrelated group may interfere with the performance of a given group. For instance, if a member of a given group fails, the re-

configuration of that group may affect the behavior of other non-overlapping groups which should not happen in the ideal situation. Due to this reason, finding appropriate mappings is a critical point in the design of a Light-Weight Group Service.

Light-Weight Group Services have been implemented before in different group based communication systems [5, 8]. Unfortunately, such services imposed restrictions on group usage as a result of changing the interface of the underlying virtually synchronous group. In a recent paper [9], we have proposed a new design for the LWG protocols that circumvents such limitations, in particular, we have shown that the Light-Weight Group Service can be implemented in a fully transparent manner. This new set of protocols has already proven advantageous in static environments, where there is large overlap among user groups.

In this paper we extend our work to dynamic environments, where mappings cannot be defined a priori and may change over time. We show the possibility to establish mappings that promote sharing and, at the same time, minimize interference. The establishment of these mappings can be automated, using heuristics applied locally at each node. Experiments implementing this approach in the Horus system [12] have shown significant performance improvements.

The paper is organized into five other sections. Related work is surveyed in Section 2. In a recent paper [9], the design of the Transparent Light-Weight Group Service is further explored, but Section 3 provides a brief description. The heuristics to support dynamic mappings are discussed in Section 4. Performance results obtained with an implementation in Horus are presented in Section 5. Section 6 concludes the paper.

2 Related work

To our knowledge, Delta-4 [8] was the first system to offer some form of Light-Weight Group Service. The Delta-4 group communication subsystem was structured as a layered architecture similar to the ISO stack. Virtually synchronous support was provided in the lower layers of the architecture, immediately on top of standard MAC protocols. Several session level groups can be mapped onto a single MAC level group, but the association was

statically defined, called a *connection* in the Delta-4 terminology.

The Isis system has extended this principle, offering a Light-Weight Group Service that supports dynamic associations between user level groups and core Isis groups [5]. 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 do not preserve the original HWG interface. In both cases, additional information is required. This requirement forces existing applications to be changed in two ways. The first change is that it prevents the LWG protocols from being used as an optional feature in a transparent manner. The second change is that it nullifies one of the most powerful features of virtual synchrony, the ability of operation without a priori knowledge of the group membership. Moreover, these previous approaches avoid the problem of finding the most appropriate mapping in a fully automated way by placing the burden on the user to locate and provide a priori knowledge of possible sources of interference.

Some systems [1] do have all groups in the system sharing some resources, such as a failure detector or an underlying ordered channel. Although these systems are implementing a static form of a Light-Weight Group Service, they do not address the problem of minimizing interference. To our knowledge, this paper presents the first LWG service that promotes resource sharing and, at the same time, minimizes interference, in a fully automated manner.

3 The transparent LWG service

3.1 Design overview

The main goal of the dynamic LWG Service is to support resource sharing by mapping several LWGs with similar membership onto a single HWG while fully preserving the original HWG interface. For added efficiency, we also construct this mapping between LWGs and HWGs in a completely automated manner. As a positive side effect of resource sharing, it is possible to decrease the latency of group operations by avoiding redundant startup procedures.

The LWG Service performs its task by managing a pool of HWGs and es-

establishing associations between LWGs and these HWGs. Every time a new LWG is created, the Service must decide whether the LWG should be associated with one of the existing HWGs (if any), or if a new HWG should be added to the pool. Whatever decision is made, the new LWG will be associated with some HWG and may 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 evolves. To compensate for these changes, the LWG Service dynamically redefines these mappings. When this happens, we say that a LWG is *switched* from one HWG to another. If, at some point, a given HWG becomes unsuitable for establishing further mappings, it is released from the pool. Thus, the pool of HWGs managed by the Service expands and shrinks over time, not only due to the creation of new LWGs, but also due to changes in membership in these groups.

The LWG service then 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 is a protocol that allows the association between a LWG and a HWG to be changed at run time.

The first task is critical to the overall design, because if no performance advantages can be obtained by mapping several LWGs onto a single HWG, the mapping and switching strategies become pointless. Due to this reason, we first concentrate on developing and evaluating the protocols that support the transparent LWG design. These protocols are described in detail elsewhere [9]. For background information, the next section provides a brief description of these protocols and of their performance.

3.2 Protocol operation

The protocols that implement the Light-Weight Group Service perform the tasks required to offer virtual synchrony (join a group, leave a group, and multicast messages in a group) and switch the mappings between LWGs and HWGs dynamically. These protocols are not tied to any particular architecture, but were designed having the Isis, Horus [13, 12] and NAVTECH [14] systems in mind. All these systems provide a virtually synchronous communication service.

Downcalls	
Name	Parameters
Join	GroupId gid, Pid pid
Leave	GroupId gid, Pid pid
Send	GroupId gid, BitArray data
HoldOk	GroupId gid
Upcalls	
Name	Parameters
View	GroupId gid, PidList view
Data	GroupId gid, Pid src, BitArray data
Hold	GroupId gid

Table 1: VS interface primitives

3.2.1 Interface

A typical interface of a virtually synchronous layer contains the following primitives, as listed in Table 1 (we denote the downcalls with the “.req” suffix and the upcalls with the “.int” suffix): `Join.req`, allows a member to join a group; `Leave.req`, allows a member to leave a group; `Send.req`, sends a virtually synchronous multicast; `View.int`, installs a new view; `Data.int`, indicates the delivery of a multicast; `Hold.int`, indicates that the traffic must be stopped temporarily (usually, when a view change in the virtually synchronous layer is in process); and `HoldOk.req`, confirms the `Hold.int` indication. `Hold.int` and `HoldOk.req` may be hidden from the user at upper layers.

The main goal of our design is to build a service that lets several user groups to share the same virtually synchronous group in a transparent manner. To accomplish this goal, the Light-Weight Group Service exports the same interface as the virtual synchrony service, as illustrated in Figure 1.

3.2.2 Flush protocol

The core of the Light-Weight Group implementation is the flush protocol, which is responsible for installing a new view. The protocol ensures that all

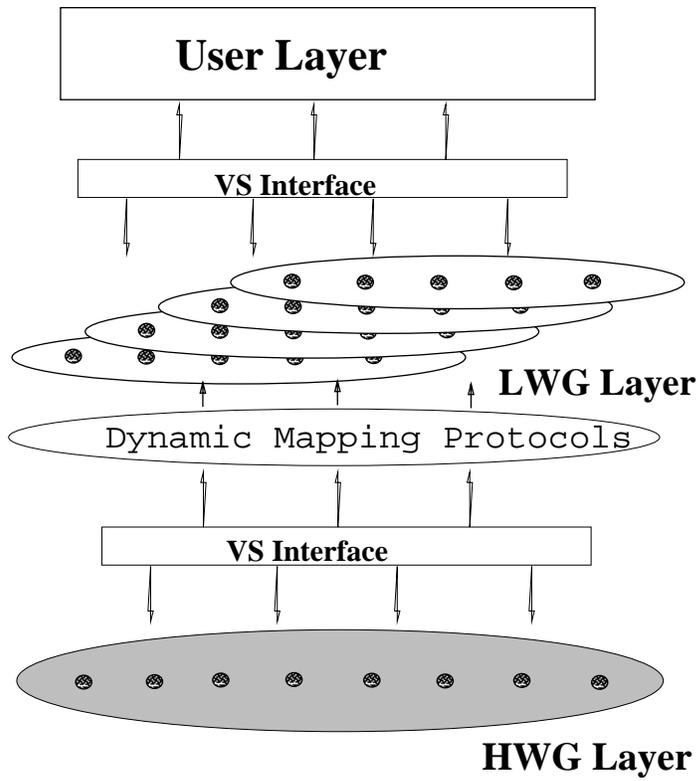


Figure 1: Light-Weight Group service interface

messages delivered to some processes in a given view are delivered to all the correct processes in that view before a new view is installed. The protocol works as follows. The oldest member of the group, the coordinator, multicasts a FLUSH message to initiate the protocol. When the FLUSH is received, the application is requested to stop sending through the `Hold.int` interrupt. When the corresponding `HoldOk.req` is received from the application, the LWG member acknowledges the FLUSH message with a FLUSH_OK. The protocol is terminated by the coordinator which sends a VIEW message as soon as a FLUSH_OK is received from every member. When the VIEW message is received, the traffic is resumed by delivering the new view through the `lwg.View.int` interrupt. In addition to the new membership of the group, the VIEW message disseminates the identity of the appropriate HWG for the next view. Thus, in our LWG Service, the flush protocol both changes the group membership and executes the switch protocol. If a member process fails or becomes unreachable while executing the flush protocol, another round of the flush protocol starts immediately, collecting FLUSH_OK replies from currently available members. Therefore, the flush protocol does not block.

3.2.3 Create/join and leave protocols

The create/join procedure consists of two main steps. In the first step, a map is established between the LWG and some HWG. These mappings need to be stored in a way that can be accessed by every process. Our protocols store mappings in an external *Name Service*. To minimize access to the name service, the joining process proposes a mapping based on its own local HWGs according to the mapping heuristics discussed in Section 4. In a single access to the name service, the joining process either commits this mapping or, in the case where the LWG is already mapped onto some other HWG, obtains the existing mapping. Additionally, if the process is not a member of the selected HWG, it joins the HWG before executing the second step. The second step consists of sending a JOIN message to all members of the HWG. When the JOIN message is received, the identifier of the joining process is added to a `joiningList` and the coordinator of the LWG triggers a flush protocol which, in turn, will install a new view.

The leave procedure is similar to the joining protocol. The process simply

sends a LEAVE message to all members of the HWG . When the LEAVE message is received, the identifier of the process is added to a `leavingList` and the coordinator of the LWG triggers a flush protocol which, in turn, will install a new view.

3.2.4 Message passing protocol

The principle of the message passing protocol is very simple. The LWG service encapsulates the LWG data in a dedicated message which is multicast on the HWG . On the recipient side, when such message is received, the `lwgId` part is examined and the data part is forwarded to the specified LWG .

A message multicast on a HWG can be performed using two main approaches. In one approach, known as selective multicasting, the message is multicast only to the relevant members of the HWG [10]. In order to be efficient, this approach requires some hardware support. In the other approach, the message is multicast to all members of the HWG and, then, each site that is not a member of the concerned LWG discards the message. This is one of the sources of interference, since memory and CPU are wasted in the processes where the message is received only to be discarded.

3.2.5 Switch protocol

Assume that a given LWG , `lwgId`, needs to be switched from one HWG , `hwgFrom`, to another HWG , `hwgTo`. The switch protocol is initiated by some process member of `lwgId`. In order to inform other members of `lwgId` of the start of the switching procedure, it multicasts an OPEN message on `hwgFrom`. When this message is received, all members of `lwgId` check if they are already members of `hwgTo` and, when they are not, join this group. When all members have joined `hwgTo`, the execution of the flush protocol is triggered. The protocol installs a new view and commits the new mapping. When a switch occurs, the name service is informed of the new mapping so that further joins are directed to the appropriate HWG (see [9] for a discussion of how concurrent joins and switches are handled).

3.2.6 Failure handling protocol

The basic failure handling protocol is quite simple because most of the complexity is handled by the virtually synchronous service. Whenever a failure is detected by a HWG, a `Hold.int` is generated in order to stop the traffic flow. This interrupt must be multiplexed to all LWGs mapped onto that HWG. The Light-Weight Group Service waits for an acknowledgment from every LWG (in-transit messages can still be sent or received) and then acknowledges the `Hold.int` interrupt. Finally, when a new view is installed in the HWG, the failed processes are removed from the views of all mapped LWGs.

3.3 An Implementation in Horus

3.3.1 Horus overview

Horus [13] is a group communication system which offers great flexibility in the properties provided by protocols. It uses virtually synchronous protocols to support dynamic group membership, message ordering, synchronization and failure handling.

In the Horus architecture, protocols are constructed dynamically by stacking microprotocols, which support a common interface. Each microprotocol offers a small integral set of communication properties, and Horus implements them as different layers.

Horus provides a large set of microprotocols. The following layers are related to our design of the Light-Weight Group Service. The COM layer provides the Horus interface over other low-level communication interfaces which include IP, UDP, ATM, the x-kernel [7] and a network simulator. The NAK layer provides reliable FIFO unicast and multicast. The FRAG layer implements fragmentation and reassembly of messages. The MBRSHIP layer guarantees virtual synchrony. In order to do failure detection and offer reliable FIFO communication, the NAK layer has each member periodically multicasts "status" background messages in the group it is residing. These "status" messages add load to the system as the number of groups in the system increases. A major objective of our LWG Service design is to eliminate redundant background traffic as much as possible.

3.3.2 Performance with identical LWGs

To evaluate the effectiveness of the LWG Service, we conducted a number of tests on n identical four-member LWGs. The tests used an implementation of the LWG Service in Horus and measured its impact on different operations such as join, leave, data transfer and recovery from crashes. These tests are relevant because they show the improvements obtained when maximum resource sharing is achieved, that is, when all LWGs are mapped onto a single HWG. For self-containment, we reproduce the results for data transfer and failure recovery (the remaining results can be found in [9]).

To evaluate the effect of LWGs on failure recovery, we conducted the following test: a given process, member of n identical four-member groups, crashes and forces these n groups to reconfigure. The recovery time, measured between the detection of the failure and the installation of a new view is presented in Figure 2. When the LWG service is used, all groups share the same recovery procedure and the recovery time is almost constant, while it shows a non-linear increase in the HWG test. To evaluate the impact of LWG on data transfer, we measured one-way latency when one member is multicasting 10-byte messages in one of the n groups. From Figure 2 we see that the LWG figure stays constant at 1.25 milliseconds, while the HWG figure increases dramatically to reach 2.9 milliseconds as n increases to 200.

These results show that the resource sharing promoted by the LWG approach offers clear performance advantages. These experiments were done in an environment where LWGs fully overlap. In this paper, we extend these results to more complex topologies and we show that, due to interference, mapping all LWGs onto a single HWG is not always the optimal solution. The next section discusses how better mappings can be achieved in the general case. .

4 Dynamic mapping

In the type of systems we are targeting (like Isis, Horus, or NAVTECH), when a process joins or creates a group, it is not required to know in advance its future membership. Actually, in most cases this membership cannot be known in advance, as it often depends on run-time parameters like number and location of users, load, occurrence of faults and so on. Thus, the LWG Service

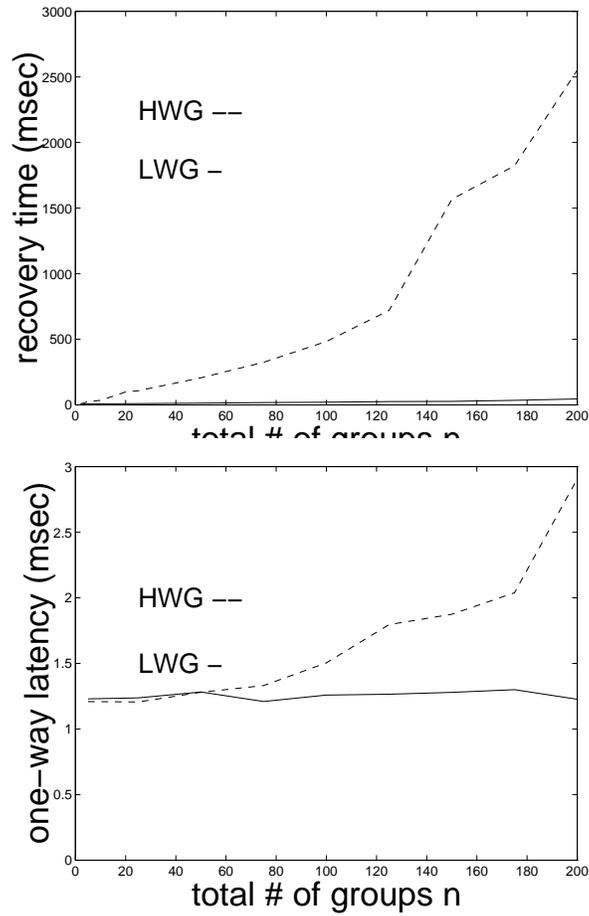


Figure 2: Recovery from crashes and data transfer

must be able to operate without this information, using heuristics to find the most appropriate mappings between LWGs and HWGs.

4.1 The problem of interference

When all LWGs have the same membership, good results are always achieved when they are mapped onto a single HWG. This begs the question whether a single HWG is always a good solution. The answer is no because two LWGs may interfere with each other if they are mapped onto the same HWG.

Usually insignificant when the LWGs have the same membership, this interference becomes increasingly disturbing as the membership differs and becomes clearly disadvantageous when the LWGs do not overlap. We can present some potential sources of interference:

The first source of interference lies in the same FIFO channel shared by all the LWGs when they are mapped onto the same HWG . All the LWGs also share the same set of physical resources the FIFO channel is built on. This, at least theoretically, reduces parallelism in the system: when a message loss from one LWG occurs, it can delay the delivery of messages from other LWGs . Today's technology rarely results in message losses due to bit errors, but losses due to congestion are still common. With Quality of Service being offered in the next generation of transport protocols such as RSVP [15] and IPv6 [4], we expect fewer and fewer message losses. Therefore, the interference from using the same FIFO channel is almost negligible in the future.

A second source of interference happens when the failure of a HWG member disturbs the operation of the HWG and the LWG (s) mapped onto it. Upon failure, the HWG usually goes through a flush procedure to enforce virtually synchronous properties (this procedure was discussed in section 3.2). Whenever the membership of the LWG does not exactly match the membership of the HWG , the failures of processes which are not its own members may disturb the operation of the LWG .

Finally, if no selective addressing is available, that is, if it is impossible to send messages to subsets of the whole HWG , all messages regarding a given LWG will be broadcast on the HWG . This means that when the membership of the HWG is a superset of the membership of the LWG , some processes will have to spend resources (CPU and memory) discarding messages not addressed to them. This is a severe source of interference because the number of messages a receiver can process per unit time is limited. This type of interference is even more severe when the members of disjoint LWGs are located in separate sub-networks. In this scenario, mapping both LWGs onto the same HWG would overload the network with traffic that could be localized to each sub-network.

Thus, the task of mapping LWGs onto HWGs addresses two conflicting goals: increasing resource sharing and minimizing interference.

4.2 The mapping problem

An optimal mapping exists for a given set of LWGs that balances the twin goals of increasing resource sharing and minimizing interference. The determination of this optimal mapping depends not only on the criterion used to evaluate the mapping (throughput, latency, failure-recovery time, etc.), but also on several operational parameters, such as the protocols being used, processing power, available memory, hardware architecture, and the network technology and topology, etc. In the rest of this paper, we will concentrate on general criteria that we believe effective in most architectures. Nevertheless, to provide some insight on the general criteria, we summarize the analysis of the mapping problem for a specific architecture (the Horus system) from two different perspectives, resource sharing and interference caused by data traffic.

In the Horus system, failure detection is performed by having each group member multicast to every other member a “status” report background message periodically (in the current configuration, every 2 seconds). Experiments show that the bottleneck for handling the background messages is the receiver buffer size [6]. Processors do not have enough input buffer space to handle all the input messages in the short amount of time, therefore, some messages are dropped from the input buffers. As a result, an important goal of resource sharing is to minimize the number of these background messages. Consider a pair of overlapping LWGs, (lwg_1, lwg_2) of size $(n_1 + k)$ and $(n_2 + k)$ respectively where $k > 0$ is the size of their overlap. If each LWG is mapped onto a different HWG (with membership identical to that of the LWG), $F_a = (n_1 + k)^2 + (n_2 + k)^2$ message interrupts are generated in the system every period. If both groups are mapped onto the same HWG, this number is $F_b = (n_1 + n_2 + k)^2$. Effective resource sharing occurs when $F_b < F_a$, that is, when $k > \sqrt{2n_1n_2}$. Another way to express this result is that a poorly chosen resource sharing policy may induce, only due to the failure detection mechanism, an overhead of $|F_a - F_b|f = |k^2 - 2n_1n_2|f$ additional message interrupts per unit time, where f is the frequency at which the failure detector runs.

Consider now the effect of interference. In Horus, selective addressing is not available. This means that a multicast on a given LWG is received by all members of the underlying HWG. For simplicity, assume the same scenario

as above and assume that every member multicasts periodically a message to the each HWG it belongs to. If each LWG is mapped onto a different HWG, each process only receives the messages addressed to it. For example, a process in lwg_1 only would receive $(n_1 + k)$ messages periodically, while a member of both lwg_1 and lwg_2 would receive $(n_1 + n_2 + 2k)$ messages, etc. If both groups are mapped onto the same HWG, every member would receive $(n_1 + n_2 + k)$ messages periodically, regardless of its affiliation. This overhead caused by interference may cancel the savings induced by resource sharing. The above analysis applies to any pair of overlapping LWGs (lwg_1, lwg_2). Either they are mapped onto two different HWGs with sizes being $(n_1 + k)$ and $(n_2 + k)$ respectively, where k is the overlap of the HWGs; or they are mapped onto one HWG of size $(n_1 + n_2 + k)$.

Of course, other effects could be taken into account. For instance, one could examine the number of messages exchanged during failure recovery, or the network load distribution according to the topology. However, these considerations are not addressed here since the focus of this paper is not the quest for the optimal mapping for a given system. Instead, the focus is to show that better performance can be achieved when resource sharing and interference are simultaneously taken into account. Later in the performance section, we show that even applying some generic rules can be effective.

4.3 Create mappings

When a LWG group is created, a mapping needs to be established. Unfortunately, the future membership of either the new LWG or of the existing HWGs cannot be foreseen (the membership of a HWG is forced to grow with the membership of the mapped LWGs). As a result, the mapping decision is of a heuristic nature, and consequently, prone to non-optimal results. Naturally, heuristics can be tuned for specific applications. We concentrate on general-purpose heuristics. Two main approaches can be followed when establishing a mapping for a LWG which is being created. They are described below.

The *pessimistic* approach assumes the membership of the new LWG will be extremely different from that of other running LWGs, and creates a new HWG. If the assumption is proven later to be incorrect, one can try to *switch* the LWG to a more appropriate HWG. The disadvantage of this approach

is that the heavier operation, creating a new HWG , is always executed by default.

The *optimistic* approach 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 . This approach has the advantage of performing by default a less expensive operation. Furthermore, the expensive operation of joining a HWG , if necessary, can be executed at a less critical point of the application execution-path (for instance, using moments of reduced communication).

Due to its advantages, we have followed the optimistic approach. The mapping strategy works as follows:

optimistic mapping rule: *when a LWG is created, the LWG Service maps the LWG onto an existing HWG with larger membership, that is, a HWG with higher probability of including future members of the LWG . If several HWGs match this criterion, the HWG with less LWGs already mapped onto it is selected which minimizes the shared channel disadvantage.*

4.4 Adaptive strategies

The mapping rule tries to promote appropriate mappings. However, due to the lack of information about the future, some of the mappings done at group creation time will later reveal to be disadvantageous. For instance, two non-overlapping LWGs might be mapped on the same HWG . In extreme cases, the mapping heuristic could lead to the existence of a single huge HWG in the system on which all LWGs were mapped. To prevent such cases from occurring, our approach includes the use of corrective measures. These 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:

- 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. Also, some architectures contain a limited number of hardware multicast addresses; it is possible to assign one of such addresses to each HWG if the number of HWGs is small.

- To minimize interference, a LWG should be mapped onto a HWG with a similar membership.
- 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, a HWG with no LWG mapped onto it should be deleted.

These guidelines can be applied in several ways. One is to rely on some central entity to determine a better configuration based on a snapshot of the existing mappings. After determining the new configuration, this central entity instructs each process about which LWGs should be switched. This solution is inherently not scalable.

Another approach relies exclusively on local heuristics to make switching decisions. This requires special care to ensure that local heuristics make the system converge to some stable configuration. It is also more difficult, if not impossible, to reach the optimal configuration using only local information. Nevertheless, this approach is scalable and allows non overlapping sub-systems to operate with total independence. Due to this reason we have experimented a number of local heuristics.

The merge, switch, shrink algorithms presented in Figure 3 are executed at every process and are based on comparing the membership of all LWGs and HWGs that are known to that process.

4.5 Instability, convergence and switching overhead

Poorly chosen local heuristics leads 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.

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 $\text{sizeof}(g_1) \leq \text{sizeof}(g_2)/k_m$.

closeness: given $g_1 \subseteq g_2$, g_1 and g_2 are close enough to each other
iff $\text{sizeof}(g_2) - \text{sizeof}(g_1) \leq \text{sizeof}(g_2)/k_c$.

Merge rule (for some configuration parameter k_m)
Considering a LWGs pair (lw_1, lw_2) with (hw_1, hw_2) as their
underlying HWG pair, where $\text{sizeof}(hw_1) = n_1 + k$,
 $\text{sizeof}(hw_2) = n_2 + k$ and $\text{sizeof}(hw_1 \cap hw_2) = k$.
if $\neg ((hw_1 \subseteq hw_2 \wedge hw_1 \text{ is a minority of } hw_2)$
 $\vee (hw_2 \subseteq hw_1 \wedge hw_2 \text{ is a minority of } hw_1))$
 $\wedge (k > \sqrt{2n_1n_2})$ **then**
merge hw_1 and hw_2 into a single hw ;
fi

Switch rule
Considering a LWG lw_1 with hw_1 as its underlying HWG .
if (lw_1 is a minority of hw_1) **then**
if ($\exists hw_x$ with membership close enough to lw_1) **then**
switch lw_1 to hw_x ;
else
create a hw_{new} with membership identical to lw_1 ;
switch lw_1 to hw_{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 3: The local algorithms

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 in a way that a significant change in the membership is required to define a new mapping. Specifically, in our tests we have used $k_m = 4$, $k_c = 4$. 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 switches when groups are being created or deleted, the heuristics are applied periodically with a relative large period. This period is configurable; in our experiments, we have run the heuristics once every minute. This also makes the overhead of executing the heuristics and running the switch protocol negligible.

Although these heuristics were used with success in our experiments, we must emphasize that the heuristics themselves were not the main goal of our research. Our major concern was to implement and test the mechanisms that allow mappings to be redefined at run-time and actually show that performance improvements can be achieved if such heuristics are available.

5 Performance results

We tested the performance in Horus on a system of 8 SUN Sparc 10 workstations running SunOS 4.1.3, connected by a loaded 10M bps Ethernet. The low level protocols used were UDP/IP with the Deering multicast extension. We used two sets of n user groups where each group within a set has identical membership. Set A contains user groups a_1 to a_n , and set B, b_1 to b_n .

We define *average multicast latency* as the average of the latencies of multicast from each member in user group a_1 and each member in user group b_1

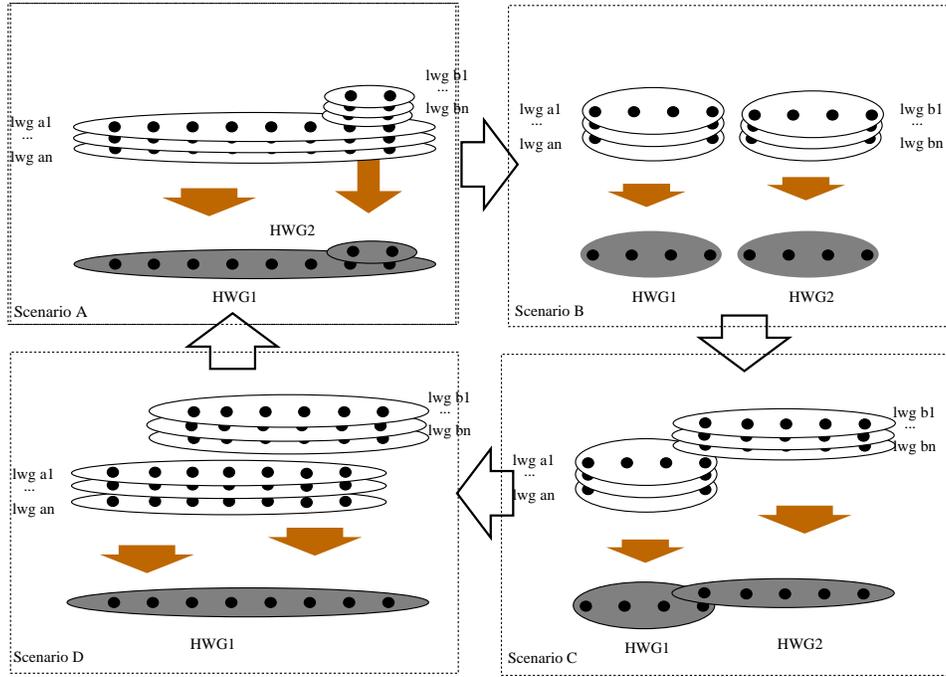


Figure 4: Test scenarios and mappings

to its respective group. The latency associated with a member was measured when it is the only member multicasting 10-byte messages in the system. This latency was averaged over 5000 tests.

To measure failure recovery time, we conducted the following test. Process p_a , a member of n user groups a_1 to a_n , crashes and a_1 to a_n are re-configured. The crash recovery time R_a for groups in set A, measured at their coordinator is the time between the detection of the failure and the installation of a new view. This crash recovery time was averaged over 50 tests. Set B was tested in a similar manner where process p_b , a member of n user groups b_1 to b_n crashes, resulting in the reconfiguration of b_1 to b_n . R_b is the crash recovery time for groups in set B.

We define *average crash recovery time* to be $R = (s_a/s) \times R_a + (s_b/s) \times R_b$, where s_a and s_b are the group sizes for groups in set A and B respectively, and $s = s_a + s_b$. This weighted average reflects the effect of the mapping on each member of the user groups.

In order to evaluate the performance of the Dynamic LWG Service, we have the system evolve through a series of scenarios, each corresponding to a different LWG membership setting. The four test scenarios A-D, illustrated in Figure 4, cover the most representative combinations of LWG membership, highlighting the advantages of LWGs. When the scenario changes, switching decisions are made according to local heuristics and the switch protocol is invoked as described in section 3.2.5. For a more detailed description of this protocol, see [9].

In scenario *A*, LWGs a_1 to a_n are mapped on HWG1. LWGs b_1 to b_n are contained in LWGs a_1 to a_n . Since the membership of b_1 to b_n is just a minority of HWG1, they are dynamically mapped on a separate HWG (HWG2). In scenario *B*, LWGs a_1 to a_n become disjoint with LWGs b_1 to b_n , so the mapping remains unchanged and the membership of the HWGs follows the membership changes of the mapped LWGs. In scenario *C*, there is a partial overlap of LWGs but not enough to merge them in a single HWG. In scenario *D*, the partial overlap of LWGs becomes large enough to trigger the merge rule, therefore all the LWGs are mapped onto one HWG (HWG1), and HWG2 is destroyed. Finally, the system goes back to Scenario *A* where both HWGs are used to support the LWGs. We have measured the average multicast latency and average crash recovery time for each scenario in three different configurations:

- without the LWG Service, where each user group has its own HWG .
- with a simple static LWG Service, where all LWGs are mapped on the same HWG .
- with the Dynamic LWG Service, where mappings are obtained using the optimal mapping rules described in previous sections.

Figure 5 shows the results for all scenarios and for all configurations. We will discuss each scenario in turn.

In scenario *A*, LWGs in set B each have two members, but they are mapped to an 8-member HWG in the static case. This non-optimal mapping increases the interference between the LWGs in set B and the underlying HWG which results in the average multicast latency of 3.25 milliseconds. In the optimal mapping obtained from the dynamic configuration, groups in

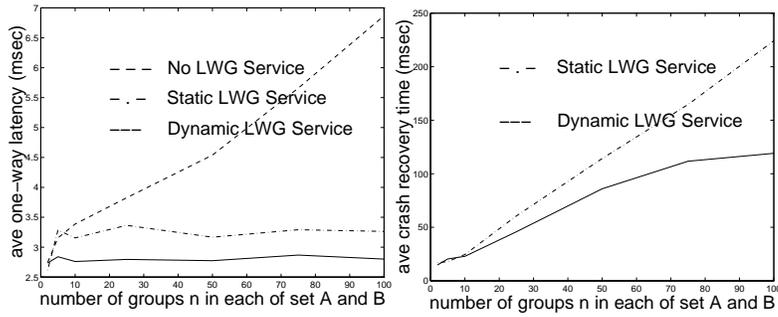
set B are mapped onto a two-member HWG2. The latency reduces to 2.75 milliseconds which is 18% lower than that of the static configuration. The average crash recovery time in both cases increases linearly with number of groups n . The slope for the dynamic mapping is less steep than that of the static mapping. The recovery time for the all-HWG case is too big to fit in the figure, as when $n = 100$, the recovery time reaches 5 seconds.

In scenario B, all the LWGs are of size 4, but, in the static case, they are mapped onto an 8-member HWG. This non-optimal mapping significantly increases the interference between the LWGs and the underlying HWG, resulting in the latency being 2.25 milliseconds. The interference is extremely large in this setting. The latency of the static mapping is even worse than without a LWG service, where all user groups are mapped onto a separate HWG. In the optimal configuration, LWGs in set A and B are mapped onto HWG1 and HWG2 respectively. The reduced interference brings the latency down to 1.25 milliseconds, a number 80% lower than the static case. In all cases, the crash recovery time increases linearly with number of groups n . Specifically, the dynamic mapping outperforms the static mapping and either mapping performs better than no LWG service at all.

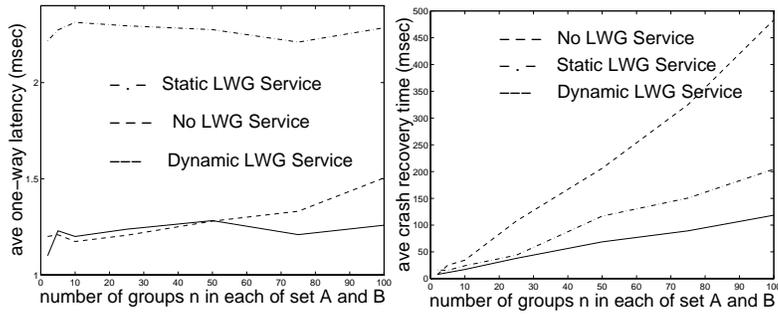
Scenario C is similar to scenario B, except each group in set B has 5 members. When all the LWGs are mapped onto the same HWG in the static case, the latency is 2.45 milliseconds. When LWGs in set A and B are mapped onto HWG1 and HWG2 respectively, the latency is 1.4 milliseconds, which is 75% lower than the static configuration. As in previous scenarios, the crash recovery time increases linearly with the number of groups n in all cases. The slope for the dynamic mapping is less steep than that of the static case. When no LWG service is used we observe the largest recovery time: it reaches 2.2 seconds when $n = 100$.

In scenario D, the overlap between LWGs in set A and set B is big enough to trigger the merge algorithm, and all groups are mapped on a single HWG. Naturally, in this case, it does not make sense to compare the static and dynamic cases, since the configurations are the same. To discuss the impact of the merge algorithm, we show the performance before and after its execution. Before the merge happens, LWGs in set A are mapped onto one HWG of size 7, and LWGs in set B are mapped onto another HWG of size 6. For most n , the latency is only 3% lower after merging, i.e. 2.35 milliseconds versus 2.425 milliseconds. This small improvement is expected because only two HWGs are

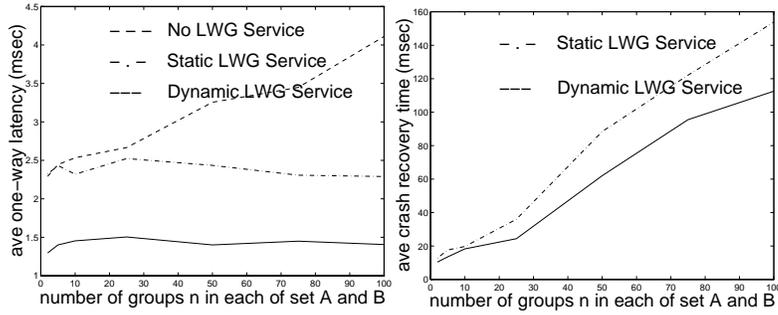
Scenario A



Scenario B



Scenario C



Scenario D

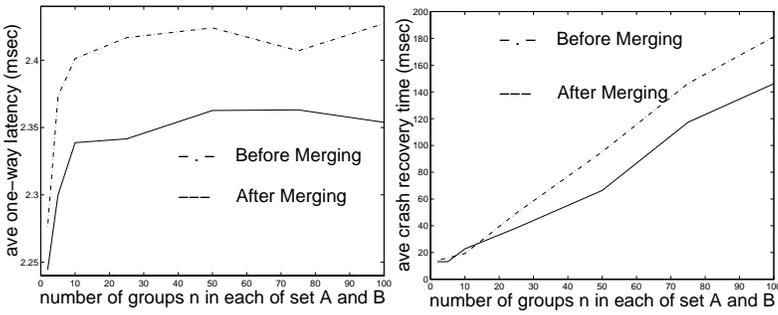


Figure 5: Test scenarios

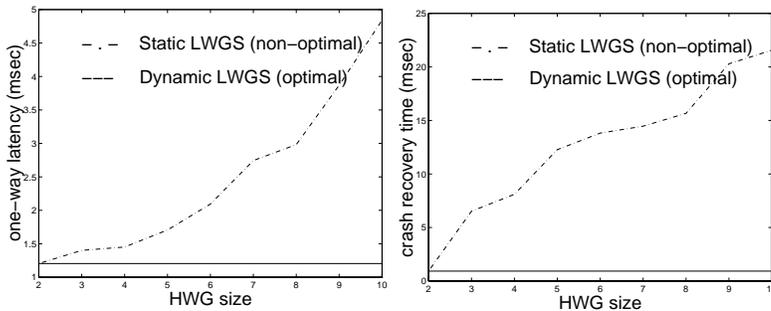


Figure 6: Switch test

merged and resource sharing is limited. If a LWG service is not used, however, the latency increases sharply as n increases, reaching 62 milliseconds when $n = 100$ (Note: it is not presented in the figure because of the scale). Again, the crash recovery time increases linearly with number of groups n in all cases. The slope after merging is less steep than before merging. Once more, the largest recovery time is obtained without any LWG service; it reaches 3.7 seconds when $n = 100$ which is also not shown because of the scale.

Finally, to illustrate the effect of interference from improper mappings, we conducted the following test. In the first non-optimal setting, we build a two-member LWG, and map it onto a HWG of size varying from 2 to 10. In the second optimal setting, this two-member LWG is mapped onto a HWG with size 2. The results are illustrated in Figure 6 and one can clearly see how interference affects the performance of the system.

6 Conclusions and future work

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. Previous work has shown that this technique is highly effective when mappings can be statically defined.

In this paper we extend this approach to dynamic environments, where mappings cannot be defined a priori and may change over time. We have shown that it is possible to establish mappings that promote sharing and, at

the same time, minimize interference. The establishment of these mappings can be automated, using heuristics applied locally at each node. Experiments using an implementation in the Horus system showed three results: i) a simple static LWG service offers better performance than no LWG service at all; ii) a dynamic service which avoids interference can, significantly improve the effectiveness of this service; iii) these gains can be achieved even using some general purpose heuristics. For future work, we intend to test the effectiveness of our heuristics in other systems, such as NAVTECH.

Acknowledgments

We would like to thank Ken Birman, Andrew Feng, Roy Friedman and the anonymous referees for their many helpful comments and suggestions.

References

- [1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proc. 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.
- [2] K. Birman and T. Joseph. Exploiting replication in distributed systems. In Sape Mullender, editor, *Distributed Systems*, pages 319–366. ACM Press Frontier Series, 1989.
- [3] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing With the ISIS Toolkit*. Number ISBN 0-8186-5342-6. IEEE CS Press, March 1994.
- [4] D. Comer. *Internetworking with TCP/IP (3rd Edition)*. Prentice Hall, 1995.
- [5] 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.
- [6] K. Guo, W. Vogels, and R. van Renesse. Structured virtual synchrony: Exploring the bounds of virtually synchronous group communication. In *Proceedings of the 7th ACM SIGOPS European Workshop*, September 1996.
- [7] N. Hutchinson and L. Peterson. Design of the x-kernel. *ACM Computer Communication Review*, 18(4):65–75, August 1988.

- [8] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.
- [9] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Grade, 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, October 1996.
- [10] L. Rodrigues, P. Veríssimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA, May 1993.
- [11] 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.
- [12] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, pages 269–283, Seattle, Washington, April 1992.
- [13] R. van Renesse, K. Birman, and S. Maffei. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.
- [14] P. Veríssimo and L. Rodrigues. The NavTech large-scale distributed computing platform. Technical report, FCUL/IST. (in preparation).
- [15] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new Resource ReSerVation Protocol. *IEEE Network Magazine*, pages 8–18, September 1993.