A Dynamic Light-Weight Group Service*

Luís Rodrigues Katherine Guo

and

António Sargento Robbert van Renesse Brad Glade

Paulo Veríssimo Kenneth Birman

Abstract

The virtual synchrony model for group communication has proven to be a powerful paradigm for building distributed applications. Implementations of virtual synchrony usually require the use of failure detectors and failure recovery protocols. In applications that require the use of 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 proposes a new design for the Light-Weight Group protocols that enables the usage of this service in a transparent and dynamic manner. As a test case, the new design was implemented in the Horus system, although the underlying principles can be applied to other architectures as well. The paper also presents performance results from this implementation.

*Selected section of this report were published in the Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, Canada, October, 1996. 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 [1, 2, 12] 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 with 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, implementations require the use of failure detectors and the execution of agreement and ordering protocols. Naturally, these components consume some amount of system resources, such as bandwidth and processing power. Although the impact of these services in the overall system performance is usually small, there are opportunities for optimization when several groups have a large percentage of common members, because these groups can share common services. Such opportunities appear in many applications [6, 11], in particular when object-oriented programming styles are used [9, 14]. For instance, a parallel application programmed using an distributed object memory can create hundreds of groups with similar membership [3].

A technique that allows the previous type of optimization consists of mapping several user level groups onto a single virtually synchronous group. 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 in this context a Heavy-Weight Group (HWG). A service that maps LWGs onto HWGs is usually called a *Light-Weight Group Service.*

The LWG paradigm is being used in several real-world applications. INFS, a reliable network file system built on the Isis system, uses this paradigm by associating LWGs with replicated files. In this 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 LWG paradigm lends itself well to this application as the large number of files amortized over a small set of file replication servers cause significant sharing of HWGS between LWGS. This setting can be generalized to apply to any application which multiplexes many object groups over a smaller set of processes or machines. In recognition of this, the paradigm is supported in the Orbix+Isis product from Isis Distributed Systems, and Iona Technologies Ltd. Here LWGS play a key role in reducing the costs of object groups by amortizing many light-weight membership changes over a smaller number of HWGS.

Light-Weight Group Services have been implemented before in different group based communication systems [6, 11]. Unfortunately, in the previous work, LWGS did not preserve the exact interface of the underlying virtually synchronous group, imposing restrictions on group usage. This paper proposes a new design for the LWG protocols that circumvents such limitations, in particular, it proposes an innovative dynamic mapping approach that allows the Light-Weight Group Service to be implemented in a fully transparent manner. As a test case, the new protocols were implemented in the Horus system [13] (as a new protocol layer) but the underlying principles can be applied to other architectures (including the Isis [6] and NAVTECH [15] systems).

The paper is organized as follows. Related work is surveyed in Section 2. The design of the Light-Weight Group Service is described in Section 3, the protocols are presented in Section 4 and the heuristics to support dynamic mappings are discussed in Section 5. An implementation in Horus is presented in Section 6 and Section 7 concludes the paper.

2 Related work

To our knowledge, Delta-4 [11] 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, in a fashion similar to that of 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 that association was statically defined (such an association is 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 [6]. Still, in order to make appropriate mapping decisions, Isis LWGs require the specification of the target membership of a user group.

Neither of these approaches is transparent, in the sense that they do not preserve the original HWG interface. In both cases, it is necessary to provide additional information, and it is our belief that this additional information limits the advantages of LWGs in two ways:

- a powerful feature of virtual synchrony is that it does not require a priori knowledge of the group membership; requiring this additional information to implement the LWG protocols reduces the applicability of the system.
- having a different programming interface, not only forces existing applications to be changed, but also prevents the LWG protocols from being used as an optional feature, in a transparent manner.

In this paper, we suggest a new suite of protocols which offer the LWG abstraction transparently underneath the original HWG interface.

3 Design overview

The main goal of the dynamic LWG Service is to support resource sharing by mapping several LWGs groups with similar membership onto a single HWG in a way that fully preserves the original HWG interface. Thus, the mapping between LWGs and HWGs must be done in a completely automated manner. As a positive side effect of resource sharing, we expect to decrease the latency of group operations by avoiding redundant start-up procedures.

The LWG Service performs its task by managing a pool of HWGs and establishing associations between LWGs and these HWGs. Every time a new LWG is created, the Service must decide if this LWG should be associated with one of the already created 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 will share that HWG with other LWGs. Since the design imposes no restriction in the way the membership of LWGs changes in time, mappings that are appropriate at one point may become inefficient as the system evolves. To compensate these changes, the LWG Service allows mappings to be dynamically redefined. In such cases we say that a LWG is *switched* from one HWG to another. If at some point in time a given HWG seems to become unsuitable for establishing further mappings, it is released from the pool. Thus, the pool of HWGs managed by the Service expands and shrinks in 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) preserves the virtually synchronous interface of the HWGs to the LWG users; (ii) defines the mapping and switching policies; and (iii) invokes 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 a critical point in the overall design as, if no performance advantages can be obtained by mapping several LWGs onto a single HWG, the implementation of mapping and switching strategies becomes pointless. In the next section we present the protocols that allow us to achieve the first and third of these tasks. The mapping and switching policies are presented in Section 5.

4 Protocols

This section describes the protocols that implement the Light-Weight Group Service. These protocols perform several tasks required to offer virtual synchrony: join a group, leave a group, and multicast messages in a group. Additionally, the protocol that allows the mappings to be dynamically changed is also presented (this protocol is independent of the triggering heuristics).

4.1 Assumptions

The Light-Weight Group Service described in this paper was designed to run on any of a set of group communication architectures. Particularly, the service was designed with the Isis, Horus and NAVTECH systems in mind. All these systems provide a virtually synchronous communication service.

4.1.1 Virtual synchrony

Informally, virtual synchrony provides group membership information to each process in the form of *views* and guarantees that all processes that install two consecutive views deliver the same set of messages between these views. More formally, virtually synchronous multicast communication can be defined as follows [12]:

vs-**multicast:** *Consider a set of processes g, a view $V^i(g)$, and a message m multicast to the members of group $V^i(g)$. The multicast of message m is called a* vs-*multicast iff the following property is satisfied. If $\exists p \in V^i(g)$ which has delivered m in view $V^i(g)$ and has installed view $V^{i+1}(g)$, then every process $q \in V^i(g)$ which has installed $V^{i+1}(g)$ has delivered m before installing $V^{i+1}(g)$. The system is virtually synchronous iff every multicast is a* vs-*multicast.*

This definition imposes a total order between view changes and multicasts, but does not enforce any ordering between messages delivered in the same view. However, in this paper, we further assume that the virtually synchronous layer delivers messages according causal precedence (and that this guarantee is preserved across different groups).

The implementation of virtual synchrony requires the use of a failure detector plus the execution of some form of *flush* protocol to ensure that all messages delivered to some processes in a given view are delivered to all correct processes in that view before a new view is installed. To guarantee the termination of the flush protocol, the traffic may be temporarily stopped during the protocol execution. This may lead to a short system slow down during the execution of the view change protocol, but simplifies application design (for example, a process that multicasts a message can deliver it locally immediately without any further computation or bookkeeping). However, protocols exist that allow the continuation of the message flow during view changes [5]. It is also possible to implement a membership service that addresses explicitly the problem of network partitions [4, 10]. The implementation of the LWG service on top of such membership service is outside the scope of this paper.

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

4.1.2 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`, is invoked by a member that wants to join a group; `Leave.req`, invoked by a member that wishes to leave a group; `Send.req`, is used to send 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`, is used to confirm 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 allows several user groups to share the same virtually synchronous group in a transparent manner. Thus, the Light-Weight Group Service should export the same interface as the virtual synchrony service, as illustrated in Figure 1.

The behavior of the interface is described by the state machine illustrated in Figure 2. When the interface is not active, it is in the *Idle* state. As a response to a `Join.req`, it leaves this state to the *Joining* state where it remains until a view that contains the local process is received. From then

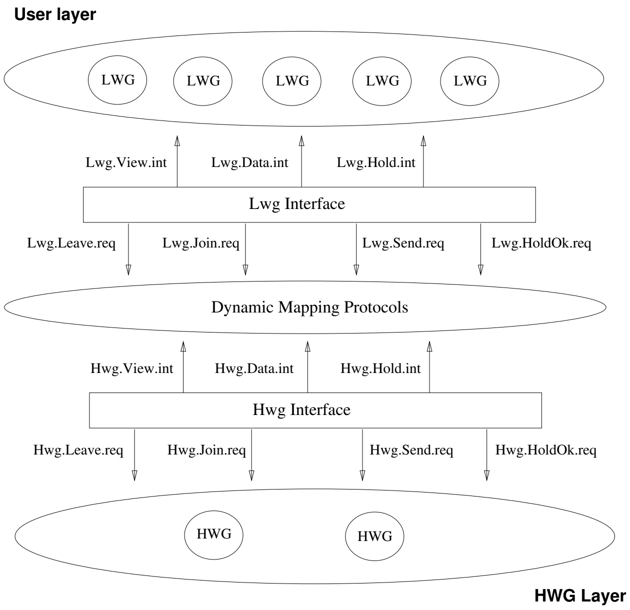

Figure 1: Light-Weight Group service interface

on, the interface is said to be in *Running* state, and can accept `Send.req` requests as well as `Data.int` interrupts. When there is the need to install a new view, the user is requested to temporarily stop sending new messages through the `Hold.int`. The interface remains in the *Holding* state until the user acknowledges this request through the `HoldOk.req`. The interface is then in the *WaitView* state, where messages from the current view can be delivered but no new messages can be sent. When a new view is received (`View.int`) the interface returns to the *Running* state. Finally, if the application wants to leave the group it issues a `Leave.req` and the interface goes to the *Leaving* state, where a view excluding the local process from the group is awaited before returning to the *Idle* state.

It should be noted that the details of the actual interface of each of the target architectures may differ. In particular, the details of the interface for the case of the Horus system will be presented in Section 6.

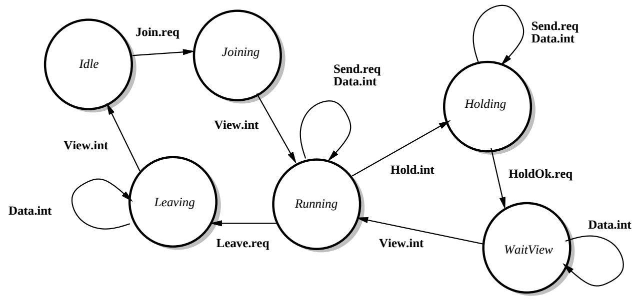

Figure 2: VS interface state machine

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: Name service interface primitives

4.1.3 Storing mappings

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. Typically, when `Join.req` is issued at some process, that process has to find out if the associated LWG is already mapped onto some HWG. In this paper we assume that mappings are stored in some external *Name Service*. The name service exports three primitives, as illustrated in Table 2, namely: `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. This last primitive is offered both to minimize the number of accesses to the name service and to prevent race conditions among processes that concurrently try to establish new mappings for the same LWG.

Note that, for availability, the name service may be replicated. A possible implementation would replicate the name service at every process, making updates expensive but read operations purely local.

4.2 Variables

The protocols use the following variables for each LWG : `lwgId`, the identifier of the LWG ; `currentHwg`, the identifier of the HWG on which the LWG is currently mapped; `nextHwg`, the identifier of the HWG where the LWG is going to be mapped in the future (usually the same as `currentHwg`, except when a switch is being executed); `currentView`, the current group view of the LWG ; `joiningList`, a list of processes that want to join the LWG ; `leavingList`, a list of processes that want to leave the LWG ; `state`, the current state of the protocol, which is one of the states showed in Figure 2; `nacks`, some protocols require an acknowledgment to be collected from every group member (the number of acknowledgments received is collected in this variable); `doFlush`, a boolean variable which is set whenever a flush needs to be performed; `coordinator`, a boolean flag which is set to true when the local process is the oldest member of the LWG .

Additionally, for each HWG , the following variables are also used: `hwgId`, the identifier of the HWG ; `currentView`, the current group view of the HWG ; `mappedLwgs`, a list of LWGs mapped onto this HWG ; `nHoldOk`, the number of `HoldOk.req` acknowledgments received. Sometimes, in order to flush the HWG , the traffic must be stopped at all mapped LWGs , `nHoldOk` is used to keep track of how many LWGs have acknowledged an `lwg.Hold.int`.

4.3 The flush protocol

The core of the Light-Weight Group implementation is the flush protocol, which is responsible for installing a new view. The protocol is illustrated in Figure 3. The protocol is initiated by the coordinator that multicasts a FLUSH message when the `doFlush` flag is set (we will later show the scenarios that trigger this condition). When the FLUSH is received, the application is requested to stop sending through the `Hold.int` interrupt (l. 307). When the corresponding `HoldOk.req` is received from the application, the LWG member acknowledges the FLUSH message with a FLUSH_OK (l. 312). The protocol is terminated by the coordinator that sends a VIEW message as soon as

```
when lwg.doFlush and lwg.coordinator and lwg.state = Running do
    lwg.doFlush := FALSE; lwg.nacks := 0;
    hwg.Send.req ( lwg.currentHwg, ⟨FLUSH, lwg.lwgId⟩);
od

when hwg.Data.int (⟨FLUSH, lwgId⟩) received do
    lwg.Hold.int ( lwg.lwgId ); lwg.state := Holding;
od

when lwg.HoldOk.req ( lwgid ) do
    lwg.state := WaitView;
    hwg.Send.req ( lwg.currentHwg, ⟨FLUSH_OK, lwg.lwgId⟩ )
od

when hwg.Data.int (⟨FLUSH_OK, lwgId⟩) received do
    lwg.nacks := lwg.nacks + 1;
od

when lwg.nacks = # lwg.curentView and lwg.coordinator do
    newView := lwg.currentView ∩ lwg.joiningList - lwg.leavingList;
    viewMess := ⟨VIEW, lwg.lwgId, lwg.nextHwg, newView⟩ ;
    hwg.Send.req ( lwg.currentHwg, viewMess );
od

when hwg.Data.int (⟨VIEW, lwgId, nextHwg, newView⟩) received do
    if local process in newView then
        lwg.currentView := newView;
        lwg.joiningList := lwg.joiningList - newView;
        lwg.leavingList := lwg.leavingList ∩ newView;
        lwg.currentHwg := nextHwg;
        if lwg.coordinator then
            ns.set ( lwg.lwgId, lwg.currentHwg ); fi
        lwg.state := Running;
    else
        lwg.state := Idle;
    fi
    lwg.View.int ( lwg.currentView );
od
```

Figure 3: Flush protocol

a FLUSH_OK is received from every member (l. 320). When the VIEW message is received, the traffic is resumed by delivering the new view through the `lwg.View.int` interrupt (l. 326). In addition to the new membership of the group, the VIEW messages disseminates the identity of the HWG that should be used during the next view (l. 331). Thus, the flush protocol is used both to change the group membership and to execute 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 can not block.

4.4 The create/join protocol

The create/join procedure consists of two main steps, as illustrated in Figure 4. In the first step, a map is established between the LWG and some HWG (l. 401). To minimize accesses to the name service, the joining process proposes a mapping based on its own local HWGs according to the mapping heuristics mentioned in Section 5. Then, in a single access to the name service it commits this mapping or, in the case where the LWG is already mapped onto some other HWG, obtains the existing mapping (l. 404). Additionally, if the process is not a member of the selected HWG , it joins the HWG before executing the second step (l. 405).

The second step consists of sending a JOIN message to all members of the HWG (l. 412). When the JOIN message is received, the identifier of the joining process is added to the `joiningList` and `doFlush` flag is activated (l. 414). The coordinator of the LWG will then trigger a flush protocol which, in turn, will install a new view.

4.5 The leave protocol

The leave procedure in Figure 5 is similar to the joining protocol. The process simply sends a LEAVE message to all members of the HWG (l. 503). When the LEAVE message is received, the identifier of the process is added to the `leavingList` and the `doFlush` flag is activated. The coordinator of the LWG will then trigger a flush protocol which, in turn, will install a new view.

```
when lwg.Join.req ( lwgId, processId ) do
      // first step
      lwg.lwgId := lwgId; lwg.state := Idle;
      lwg.currentHwg := proposeLocalMapping ();
      lwg.currentHwg := ns.testset ( lwgId, lwg.currentHwg );
      if  local process not member of hwgId then
            hwg.Join.req ( lwg.currentHwg );
            wait hwg.View.int (lwg.currentHwg);
      fi
      localMap ( lwg.lwgid, lwg.currentHwg );
      // second step
lwg.state := Joining;
      hwg.Send.req (lwg.currentHwg, ⟨JOIN, lwg.lwgId, processId⟩ );
od

when hwg.Data.int (⟨JOIN, lwgId, processId⟩) received do
      lwg.joiningList := lwg.joiningList ∪ processId;
      lwg.doFlush := TRUE;
od
```

Figure 4: The create/join protocol

```
when lwg.Leave.req ( lwgId, processId ) do
      lwg.state := Leaving;
      hwg.Send.req ( lwg.CurrentHwg, ⟨LEAVE, lwgId, processId⟩ );
od

when hwg.Data.int (⟨LEAVE, lwgId, processId⟩) received do
      lwg.leavingList := lwg.leavingList ∪ processId;
      lwg.doFlush := TRUE; // will trigger flush
od
```

Figure 5: The leave protocol

4.6 The message passing protocol

The principle of the message passing protocol is very simple. The LWG service simply encapsulates the LWG message in a dedicated ⟨DATA, lwgid, data⟩ 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 could be performed using two main approaches. In the first 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 has two disadvantages:

- it makes the multicast more expensive, since more destination sites are used than those strictly needed;
- it consumes resources to handle the received messages at those sites.

The other approach consists of using some form of selective address mechanism, which allows to multicast a message in a HWG just to a subset of all the members of the HWG. An approach similar to this was used in the Delta-4 [11] and Isis lightweight group mechanisms [6].

4.7 The 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, lwgId, hwgTo⟩ message on `hwgFrom` (l. 601). When this message is received, all members of `lwgId` check if they are already members of `hwgTo` and, in case they are not, join this group (l. 603).

When a member of the LWG detects that all members have joined *hwgTo*, it sets the variable `nextHwg` and activates the `doFlush` flag (l. 608). As in the previous cases, this will trigger the execution of the flush protocol which will install a new view and commit the new mapping. The switch protocol is presented in Figure 6.

```
when lwgId needs to be switched to hwgTo do
    hwg.Send.req ( lwg.currentHwg, ⟨OPEN, lwgId, hwgTo⟩ );

    when hwg.Data.int (⟨OPEN, lwgId, hwgTo⟩) received do
        // OPEN is received through hwgFrom
        if I am not member of hwgTo then
            hwg.Join.req ( hwgTo ); fi
    od
    when lwg.currentView ⊂ hwgTo.currentView do
        lwg.nextHwg := hwgTo; lwg.doFlush := TRUE;
    od
od
```

Figure 6: The switch protocol

4.8 The 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 (l. 700). This interrupt must be multiplexed to all LWGs mapped onto that HWG (see Figure 7). 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 (l. 706). Finally, when a new view is installed in the HWG, the failed processes are removed from the views of all mapped LWGs (l. 713).

4.9 Synchronization with the name server

When a switch occurs, the name service is informed of the new mapping so that further joins are directed to the appropriate HWG. A problem of using an external name service to keep information about the mapping between LWGs and HWGs, 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 (for instance, when a read to the name service is executed concurrently with the execution of the switch protocol). To compensate for this, all members of a HWG keep information

```
when hwg.Hold.int (hwg) do
    forall lwg in hwg.mappedLwg
        lwg.Hold.int (lwg);
    endfor
od

when lwg.HoldOk.req (lwg) do
    hwg.nHoldOk := hwg.nHoldOk +1;
    if hwg.nHoldOk = # hwg.mappedLwg then
        hwg.HoldOk.req (hwg.hwgId);
    fi
od

when hwg.View.int (hwgId, newview) do
    hwg.currentView := newview;
    forall lwg in hwg.mappedLwg do
        lwg.currentView := lwg.currentView ∩ newView;
        lwg.joiningList := lwg.joiningList ∩ newView;
        lwg.leavingList := lwg.leavingList ∩ newView;
        lwg.View.int (lwg.lwgId, lwg.curentView);
        if local process oldest in lwg.currentView then
            lwg.coordinator := TRUE;
        fi
    endfor
od
```

Figure 7: Failure handling

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. Forward-pointers are discarded based on the passage of time. Thus, we assume that when a process gets a mapping from the name service, this information is valid just for some reasonable period of time (in some sense, it works as a *lease* [7]).

4.10 Interleaving of protocols

The final protocols are slightly more complex than the ones presented in this paper due to the possible interleaving of the failure handling protocol with the remaining protocols. Figure 8 illustrates the state transition of the complete protocol.

5 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 must be able to operate with this lack of information, using heuristics to find the most appropriate mappings between LWGs and HWGs.

5.1 Potential disadvantages

In order to be effective, the mapping strategy should avoid the following potential disadvantages of having several LWGs sharing a single HWG:

- All LWGs will share the same FIFO channel and this, at least theoretically, reduces the parallelism of the system: a message losses from one LWG can delay the delivery of a message from other LWGs. In practice, since with today's technology message losses are rare events, this problem is almost negligible.

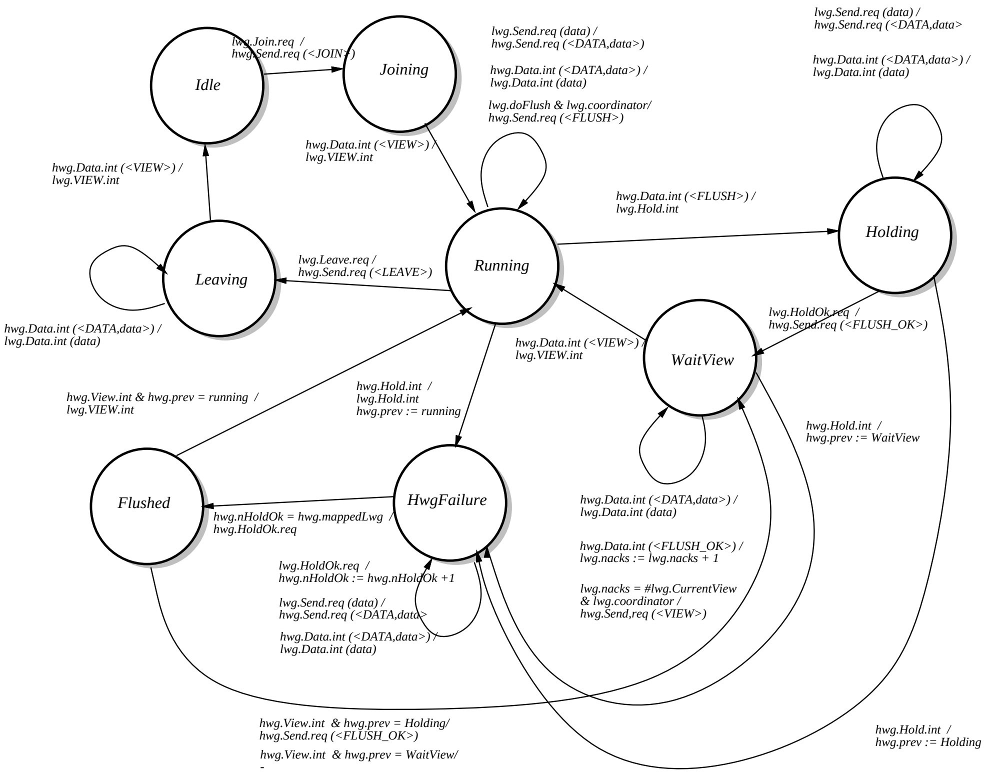

Figure 8: Light-Weight Group state machine

- If no selective addressing is available (i.e., if it is impossible to send messages to subsets of the whole HWG), all messages regarding a given LWG will have to be broadcast in 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 discarding messages not addressed to them.

- The failure of a HWG member disturbs the operation of the HWG : usually, upon failure, the HWG has to go through a flush procedure to enforce virtually synchronous properties (this procedure was discussed in section 4). Whenever the membership of the HWG does not exactly match the membership of the LWG , the operation of the LWG risks being disturbed by failures of processes which are not in the LWG.

In order to minimize these undesirable effects, LWGs should be mapped into HWGs with similar membership (ideally, with exactly the same membership). Unfortunately, when a group is created and the mapping needs to be established, the future membership of both the new LWG and of the existing HWGs cannot be foreseen (the membership of a HWG is forced to grow to match the membership of the mapped LWGs). As a result, the mapping decision is of heuristic nature, and consequently, prone to non-optimal results. Here we concentrate on general-purpose heuristics even though heuristics can be tuned for specific applications.

5.2 Policies

There are two main approaches that can be followed when establishing a mapping for a LWG which is being created:

- the *pessimistic* approach, where it is assumed that the membership of the LWG will be extremely different from that of other running LWGs , and thus, a new HWG should be created. If later the assumption is proven 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 to creat a new HWG , is always executed by default.

- the *optimistic* approach, where it is assumed that the membership of the new LWG is going to be similar to that of some other LWG already in operation. The new LWG can then be mapped onto some existing HWG and later, if the choice is proven to be inappropriate, *switch* the LWG to a more appropriate HWG. This approach has the advantage of performing by default a less expensive operation. Furthermore, if necessary, the expensive operation of joining a HWG can be executed in a less critical point of the application execution-path (for instance, using some moments of reduced communication).

5.3 Mapping rule

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 *on an existing* HWG *with larger membership, i.e., a* HWG *with higher probability of including future members of the* LWG. *If several* HWG *match this criterion, the* HWG *with the least number of* LWGs *already mapped onto it is selected (this minimizes the shared channel disadvantage).*

5.4 Moving strategies

The mapping rule tries to increase the number of 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 inappropriate. For instance, it might happen that two non-overlapping LWGs are mapped on the same HWG. In extreme cases, the mapping heuristic could lead to the existence of a single (huge) HWG in the system where all LWGs were mapped. To prevent such cases from occurring, our approach includes the use of corrective measures, which are based on the ability to change the mapping between LWGs and HWGs at run-time. This is done through a *switch* protocol, which switches a given LWG from one HWG to another HWG (the protocol was described in Section 4).

The first aspect that needs to be considered is *when* it is worthwhile to schedule a switch of a given LWG from one HWG to another. Since the more the membership of the HWG differs from that of the LWG (the HWG always contains the LWG) the more significant the disadvantages listed in Section 5.1 become. We use the following rule:

k-minority rule: *a* LWG *whose membership is a minority of the membership of the* HWG *should be switched. More precisely, the condition for scheduling the switch is, for some parameter k_m,*

NMEMB*(*LWG *)*<NMEMB*(*HWG *)/k_m,*

where NMEMB*(group) denotes the number of members in the group.*

The second aspect to consider is the selection of the new HWG for the LWG to switch to. If no other HWG exists, a new HWG should be first created to allow the execution of the switch protocol. However, if some other HWG already exists, it should be considered a potential candidate to support the LWG . Naturally, switching should minimize the disadvantages of sharing HWGs . Thus, an existing HWG should only be selected if its membership is close enough to that of the LWG being switched. We use the following rule:

k-closeness rule: *an existing* HWG *should only be eligible to support a switching* LWG *if the following inequality is true, for some parameter k_c,*

*(*NMEMB*(*HWG *)*−NMEMB*(*LWG *))*<NMEMB*(*HWG *)/k_c.*

If no HWG *exists satisfying this rule, a new* HWG *should be created to support the* LWG .

The third aspect to be considered is the timing to actually perform the switch. Ideally, switching should be done in periods of low activity of the LWG , in order to avoid disturbing the LWG traffic (the switching protocol involves flushing the group). However, in some cases, these periods may not occur with the frequency necessary to perform all the required switches. Thus, in our design we also consider that a switch protocol can be initiated when some threshold is reached.

t-threshold rule: *if the k-minority rule remains applicable for a pre-defined period of time $t_{threshold}$, and no idle moment occurs within this period, the switch protocol is initiated for that* LWG *when the threshold is exceeded despite the traffic load at that moment.*

It is also important to recognize that group membership changes can occur in bursts. In particular, during group creation and during reconfiguration (after failures) several new members may join the group in a short period of time. To avoid a cascade of switches, we should prevent the switch protocol from being started immediately after a change in configuration. To achieve this, we use the following rule:

t-stability rule: *no* LWG *should change* HWG *more than once in a pre-defined period of time t_{stable}.*

Finally, it is possible that after a LWG switch, a process finds itself a member of some HWG without having any LWG mapped on it. If this situation persists for some time, the process should leave the HWG . The following rule is used:

t-shrink rule: *a member of a* HWG *which has no* LWG *mapped on it for more than some period of time t_{shrink} must leave the* HWG *.*

Naturally, the heuristics can be improved and tuned in face of existing applications. When the software infrastructure allows it, it is may be desirable to have different heuristics for different kinds of applications.

In the previous paragraphs we have discussed what measures can be taken to minimize the disadvantages of having inappropriate mappings. Inappropriate mappings occur when the membership of the LWG is just a small subset of the membership of the HWG . The LWG Service should also make efforts to promote appropriate mappings, i.e, it should try to map all LWGs with similar membership in a single HWG .

In fact, the main goal of a LWG Service is to promote the sharing of resources and this goal is better attained if the number of HWG groups is kept low. Besides this main advantage, we can cite a number of other advantages of having a small number of HWGs . If the number of HWG is low, the search space is small and the heuristics can be applied in more efficient way. Also, in some architectures there is a limited number of hardware multicast addresses; if there is a small number of HWGs it is possible to assign one of such addresses to each HWG.

The previous heuristics do not guarantee that the number of HWGs is kept low because they do not consider as a disadvantage having several HWGs with similar membership. For instance, an application using m LWGs running on n processes could result in the following worst case. If each process creates or joins the m groups in a different order, the application might end up with m LWGs using a different HWG for each LWG, even though all the HWGs have the same membership. Also, with the evolution of the system, it is possible for groups with initially different memberships reach a point where they have similar memberships. The following rule takes these aspects into account:

k-merge rule: *two existing* HWGs, *H_1 and H_2, should be merged (by switching all* LWGs *mapped on H_2 to H_1, and discarding H_2) if the following inequality is true (assuming H_1 is bigger than H_2):*

$(\mathrm{NMEMB}(H_1) - \mathrm{NMEMB}(H_2)) < \mathrm{NMEMB}(H_1)/k_m$.

To improve the efficiency of these heuristics, the LWG Service could accept hints, in the form of probable membership or connections as in the Isis or Delta-4 systems. However, these parameters should be maintained as optional to preserve the virtual synchrony group interface.

6 An implementation in Horus

6.1 Horus overview

Horus 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 is implemented as a layer in Horus. Each layer has a set of entry points for downcall and upcall procedures denoted with the "`.req`" and "`.int`" suffixes respectively.

Horus provides a large set of microprotocols. The following are related to our design of the Light-Weight Group Service. The COM layer provides the Horus interface over other low-level communication interfaces (including IP, UDP, ATM, the x-kernel 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. The CAUSAL and TOTAL layers offer causally and totally ordered message delivery respectively.

6.2 Horus virtual synchrony protocols

The MBRSHIP layer in Horus implements virtually synchronous membership and message atomicity. During message transmission, members of the group are constantly collecting stability information of all the messages they have sent or received. A message is stable if it has been received by every member of the group. Virtual synchrony is ensured by a flush protocol that is conceptually similar to that presented in Section 4.3.

In the MBRSHIP layer, the oldest member in a view is designated as the coordinator. During a membership change, the coordinator decides which members are correct and should be included in the next view. It broadcasts a FLUSH message to the surviving members, requesting them to stop sending messages and to ignore messages from incorrect members. Upon receipt of a FLUSH, a member forwards to the coordinator its unstable messages followed by a FLUSH_OK message (these messages are point-to-point). When the coordinator has received a FLUSH_OK message from all correct processes in the current view, it rebroadcasts those unstable messages. Upon receiving rebroadcast messages, the members ignore those it has already delivered. The flush is completed after all the messages have stabilized. At this point a new view may be installed.

In our implementation, the LWG layer is put on top of the "MBRSHIP:FRAG:NAK:COM" stack. The LWG flush protocol is implemented using a coordinator based solution where the FLUSH_OK and VIEW messages carry the causal dependencies required to automatically flush data messages. Currently, all LWG messages are sent in multicast to all members of the HWG.

6.3 Performance

We conducted the performance tests for LWG in Horus 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 the Deering multicast extension. We tested n identical four-member groups using four machines with one process per machine.

We conducted three different types of tests to measure the impact of LWG Service on: (i) group membership operation, (ii) failure handling and (iii) data transfer. In these tests, every group member has the stack "LWG:-MBRSHIP:FRAG:NAK:COM" underneath it. To evaluate the effectiveness of our approach, the exactly same tests were run without the LWG layer. In the rest of the section, all the flush time measurements are taken at the coordinator. When a member joins, the flush time is measured between a `Join.req` and a `View.int`. When a member leaves, the flush time is measured between the receipt of the LEAVE message and the following `View.int`.

6.3.1 Membership operations.

To evaluate the effect of LWGs on membership operations, we measured the total flush time at the coordinator when another process joins, one by one, n groups with the same membership. We measured the flush time between the time when the coordinator receives the first `Join.req` and the last `View.int`. Figure 9 shows the flush time when a process joins as the third and fourth member. The time required to perform membership operations is a function of the number of executions of the flush protocol. Without the LWG layer, the time for joining n groups of $p-1$ members, denoted $\text{JoinTime}_{\text{HWG}}(p,n)$, can be approximately[1] expressed as follows (where $\text{Flush}_{\text{HWG}}(p)$ is the amount of time for each HWG flush when the resulting group size is p):

$$\text{JoinTime}_{\text{HWG}}(p,n) = \text{Flush}_{\text{HWG}}(p) \times n$$

[1] The measured time is smaller since there is some degree of parallelism among concurrent flushes.

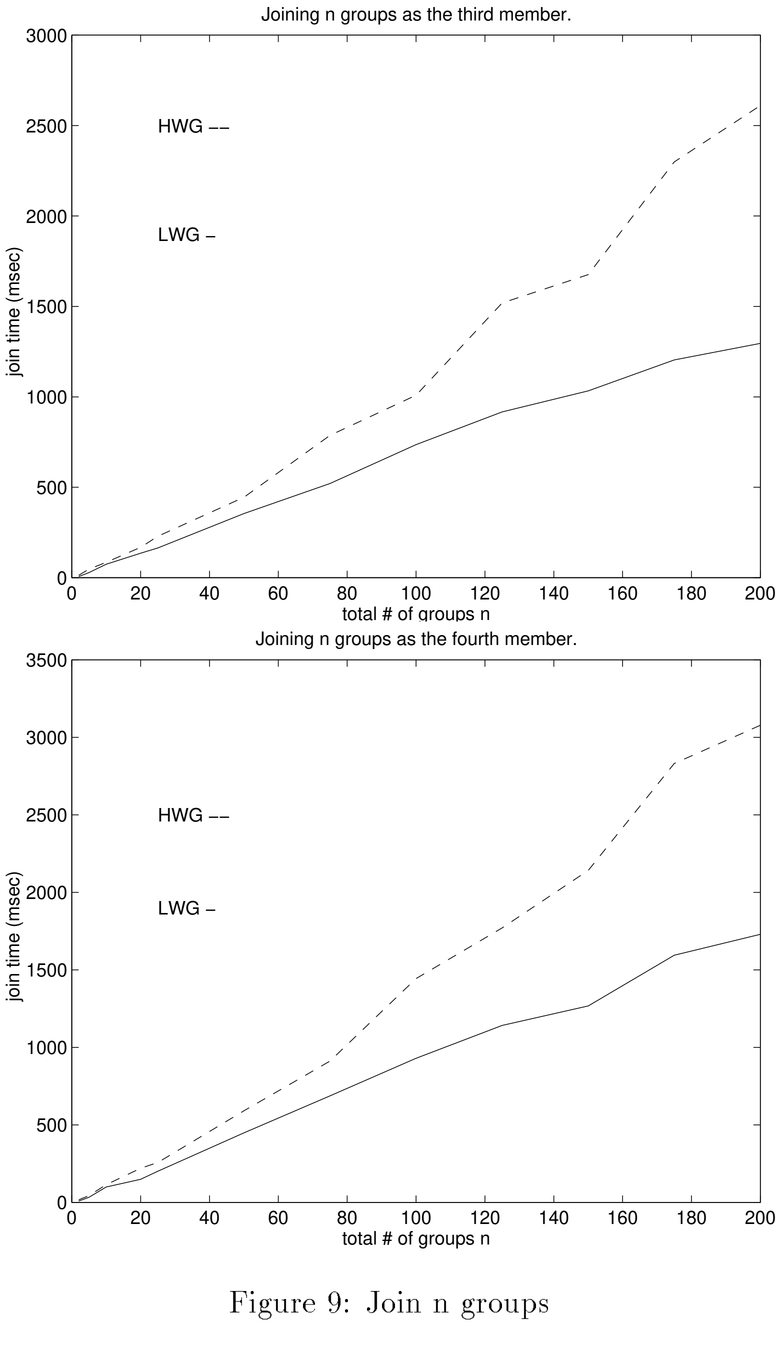

Figure 9: Join n groups

When the LWG layer is used, this time can be expressed as (where $\text{Flush}_{\text{LWG}}(p)$ is the amount of time for each LWG flush):

$$\text{JoinTime}_{\text{LWG}}(p, n) = \text{Flush}_{\text{HWG}}(p) + \text{Flush}_{\text{LWG}}(p) \times n$$

In this case, when the process joins the first of the n LWGs, it joins the underlying HWG first. As a result, the first join involves a HWG flush and a LWG flush.

In the Horus implementation, the flush process is identical for both HWGs and LWGs. In either case, the coordinator waits until it has collected FLUSH_OK messages from all other members and, as soon as the flush is done, installs a new view. The difference between $\text{Flush}_{\text{HWG}}(p)$ and $\text{Flush}_{\text{LWG}}(p)$ solely comes from the configuration of resources in the underlying layers. In the HWG approach, the MBRSHIP, FRAG, NAK and COM layers need to be reconfigured every time a new view is installed (this is performed by installing the new view in all these layers). In the LWG approach, these resources are shared and need to be reconfigured only once.

The difference between $\text{Flush}_{\text{HWG}}(p)$ and $\text{Flush}_{\text{LWG}}(p)$ is shown in Figure 10 where the flush time is measured when a non-coordinator leaves one of the n four-member groups.

The improvement of LWGs over HWGs in this case is not impressive. Still, it provides some amount of optimization whose benefit becomes non-neglegible for large number of groups. When several joins are requested in bursts, the performance can be further improved by piggybacking several independent joins and executing them in a single operation. We are planning to modify the Horus interface to allow a process to join n LWGs at once, and then build a "join-piggyback" layer that can be inserted at any point in the stack.

6.3.2 Failure recovery.

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[2] is presented

[2]In Horus, failure detection is performed at the lower layers [8].

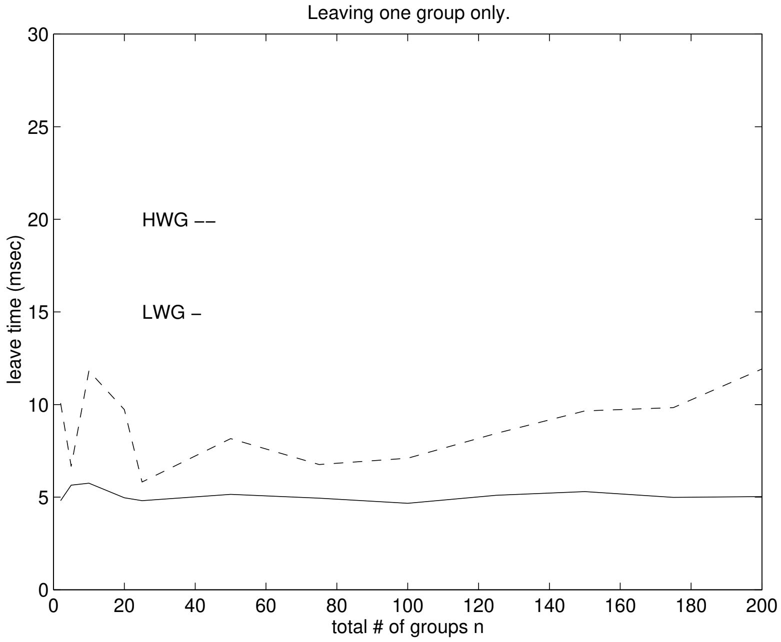

Figure 10: Leave one group

in Figure 11.

Again, the installation of a new view is preceded by a flush operation. Since a failure is notified at each of the n groups, each group starts its own flush. In the HWG test, n HWG flushes need to be run in parallel. On the other hand, the LWG layer multiplexes the flushes of all LWGs in a single flush of the underlying shared HWG. Thus, the total recovery time for HWGs shows a more than linear increase as n increases, whereas for LWGs, the total recovery time increases linearly with a very flat slop. This small linear increase is due to the fact that, in any case, all LWGs need to be notified of the group membership change. For readability, a scaled illustration of this relatively flat slop is shown in Figure 12.

6.3.3 Data transfer.

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

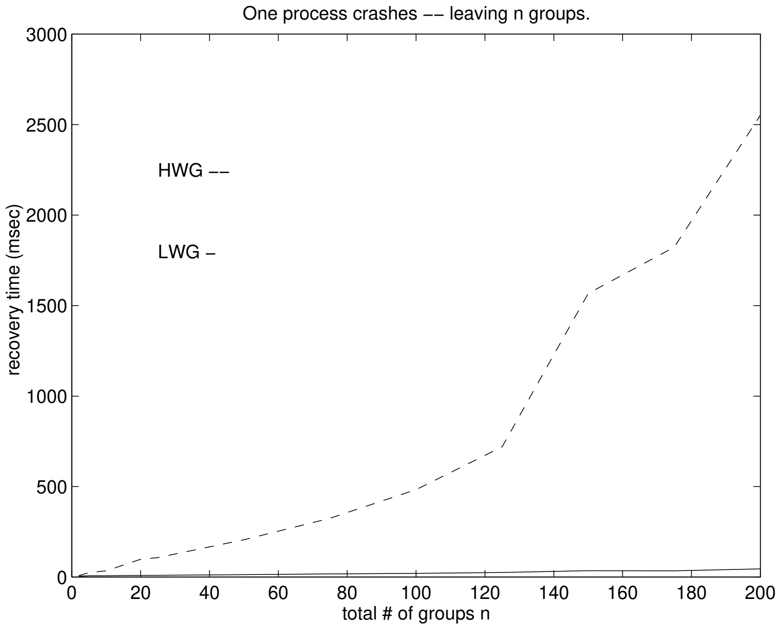

Figure 11: Recovery from crashes (comparative)

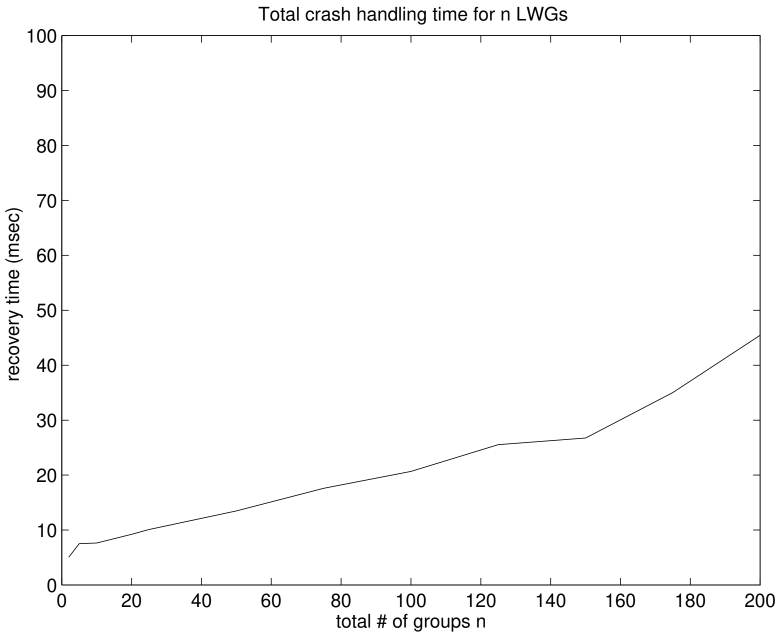

Figure 12: Recovery from crashes (Lwg)

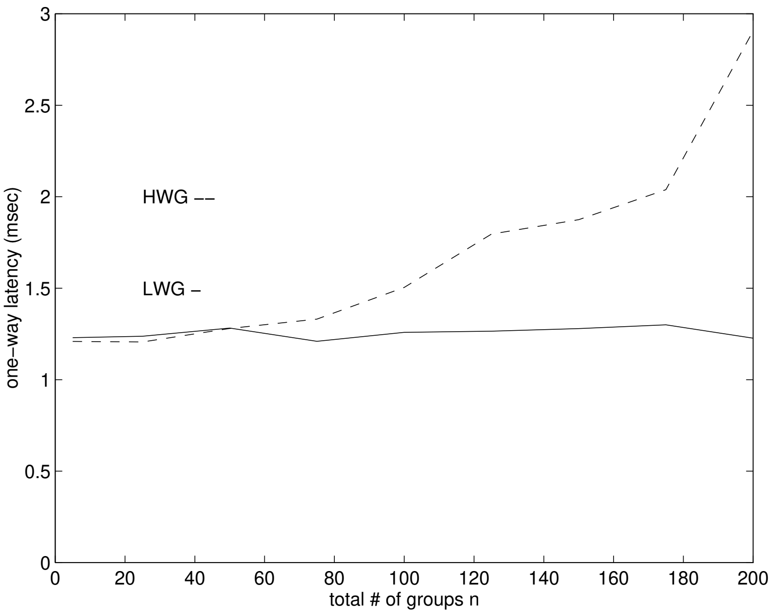

Figure 13: Data transfer

four-member groups. Figure 13 shows that up to $n = 50$, the one-way latency of the HWG test is slightly better than that of the LWG test, with the difference being 20 microseconds. After $n = 50$, The LWG figure stays constant at 1.25 milliseconds, while the HWG figure increases dramatically from 1.28 to 2.90 milliseconds as n increases from 50 to 200.

It is interesting to discuss the causes for these behaviors. In order to offer timely failure detection and reliable FIFO communication, the NAK layer in Horus has each group member multicast one "status" report background message every 2 seconds. Every member therefore receives three "status" report every 2 seconds. When there are n HWGs on each process, a total of $3n/2$ background messages need to be handled every second. Experiments show that the network bandwidth is more than enough to handle $n/2$ IP multicasts per second of small background messages even when $n = 200$. The bottleneck is the receiver processing speed [8]. As n increases, the process is not fast enough to handle all the incoming messages, therefore, it drops them from the input buffer. The resulting requests for retransmissions and retransmissions themselves add even more load to the system. This snowball effect causes the flush time and data transfer latency for n HWGs to increase dramatically with n.

These results show that the resource sharing promoted by the LWG approach offers clear performance advantages. It is interesting to observe that the significant improvements in failure recovery are not achieved at the cost of degrading other services. On the contrary, the performance of data transfer and join operations is also improved for large number of groups.

7 Conclusions and future work

In this paper we have presented a technique that promotes resource sharing among groups that have the same or similar membership. This is achieved by executing, in a fully transparent manner, a set of inexpensive protocols on top of a virtually synchronous layer. An implementation of these protocols in the Horus system has shown that this approach offers clear performance advantages. The experiments were done in a environment where the mapping between light-weight groups and heavy-weight groups remains constant over significant periods of operation. We are currently experimenting the switching heuristics (that dynamically modify this mapping) to extend these results to less stable group patterns.

References

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

[2] 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.

[3] M. Castro and N. Neves. Group communication support for parallel applications in a cluster of workstations. Technical report, INESC-IST, June 1994.

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

[5] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. In *15th Symposium on Reliable Distributed Systems*, October 1996.

[6] 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.

[7] G. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, Litchfield Park, Arizona, December 1989.

[8] 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.

[9] O. Hagsand, H. Herzog, K. Birman, and R. Cooper. Object-oriented reliable distributed programming. In *Proceedings of 2nd International Workshop on Object-Orientation in Operating Systems*, 1992.

[10] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.

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

[12] 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.

[13] 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.

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

[15] P. Veríssimo and L. Rodrigues. The NavTech large-scale distributed computing platform. Technical report, FCUL/IST. (in preparation).