# A Transparent Light-Weight Group Service


Luís Rodrigues — ler@inesc.pt
Katherine Guo — kguo@cs.cornell.edu
Antonio Sargento — amgs@pandora.inesc.pt
Robbert van Renesse — rvr@cs.cornell.edu
Brad Glade — glade@isis.com
Paulo Veríssimo — paulov@inesc.pt
Ken Birman — ken@cs.cornell.edu



**Abstract**

Virtual synchrony, also known as view synchrony, has proven to be a powerful paradigm to build distributed applications. Informally, virtual synchrony provides to each process group membership information in the form of *views* and guarantees that all processes that install a given view have delivered the same set of messages from the previous view. Implementations of virtual synchrony usually require the use of failure detectors and failure recovery protocols.

There is a range of applications that require the use of a large number of groups with the same membership. In such applications, significant performance gains can be attained if these groups share the resources required to provide virtual synchrony. A service that maps user groups into instances of a virtually synchronous implementation is called a Light-Weight Group Service.

This paper proposes a new design for the Light-Weight Group Service protocols that circumvents some of the limitations of previous approaches. As a test case, the new protocols were 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.


## 1 Introduction

Virtually synchronous group communication [1, 2, 13] has proven to be a powerful paradigm for developing distributed applications. This paradigm allows processes to be organized in *groups* within which they exchange messages in order 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 a given view have delivered the same set of messages from the previous view.

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 or 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, 10]. 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 into 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 into HWGs is usually called a *Light-Weight Group Service*.

Light-Weight Group Services have been implemented before in different group based communication systems [6, 11]. Unfortunately, in these previous works, 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 a 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 [14] (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 and the protocols are presented in Section 4. An implementation in Horus is presented in Section 5 and Section 6 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 could be mapped into a single MAC level group, but that association was statically defined by labeling all session groups (such a label 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 decision, Isis LWGs require the specification of the target membership of an user group.

None of these approaches is transparent, in the sense that none of them preserves 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 the utilization of the LWG in two ways:

- a powerful feature of virtual synchrony is that it does not require previous 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 that implement the LWG abstraction, in the meantime, still preserving the HWG interface, such that it may be optionally used with full transparency by any application.

## 3 Design Overview

The main goal of the dynamic LWG Service is to support resource sharing by mapping several LWGs groups with similar membership into 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 change in time, mappings that are appropriate at one point may become inefficient with the system progress. To compensate these changes, the LWG Service allows mappings to be dynamically redefined. In such case, 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 to establish 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 has then three main tasks: (i) preserves the virtually synchronous interface of the HWGS to the LWGS 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 in run time. In this paper we present the protocols that allow us to achieve the first of these tasks. This is a critical point in the overall design as, if no performance advantages can be obtained by mapping several LWGS in a single HWG, the implementation of mapping and switching strategies becomes pointless (mapping and switching heuristics are discussed in another report [12]).

# 4 Protocols

This section describes the protocols that implement the Light-Weight Group Service. These protocols perform the several tasks required to offer virtual synchrony: join a group, leave a group, and multicast messages in a group. Additionally, the switching protocol is also presented. The section starts by presenting the assumptions about the underlying (Heavy-Weight) virtually synchronous service.

## 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 having in mind the Isis, Horus and NAVTECH systems. All these systems provide a virtually synchronous communication service.

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

Table 1: VS Interface Primitives

#### 4.1.1 Virtual Synchrony

Informally, virtual synchrony provides each process group membership information in the form of *views* and guarantees that all processes that install a given view have delivered the same set of messages from the previous view. More formally, virtually synchronous multicast communication can be defined as follows [13]:

*vs*-**multicast:** *Consider a set of processes $g$, a view $V^i(g)$, and a message $m$ multicast to the members of group $V^i(g)$. 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 multicast of message $m$ is called a* vs-*multicast. 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. 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 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 [4, 5]. The implementation of the LWG service on top of these weaker membership services is outside the scope of this paper.

### 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 members 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; and `HoldOk.req`, is used to confirm the `Hold.int` indication `Hold.int` requires the application to stop sending new messages when a view change in the virtually synchronous layer is in process. In this paper, we assume that the virtually synchronous layer delivers messages according causal precedence (and that this guarantee is preserved across different groups).

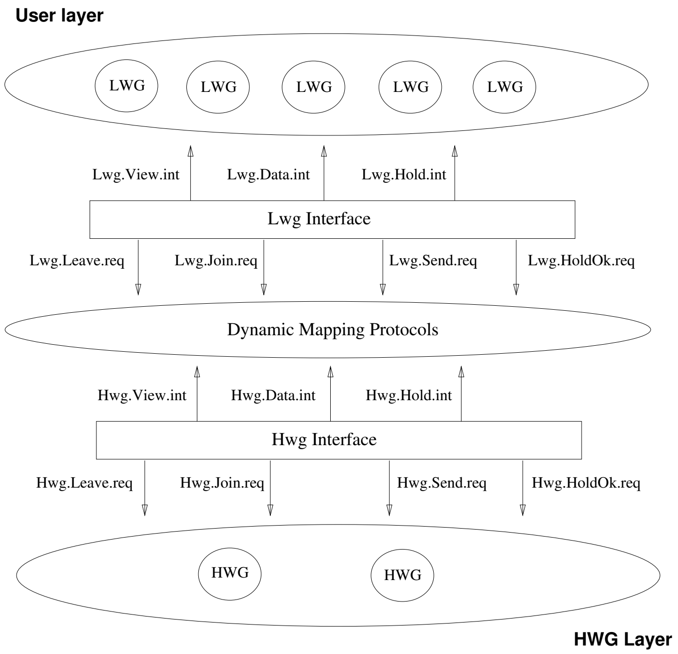


Figure 1: Light-Weight Group Service Interface

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 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.

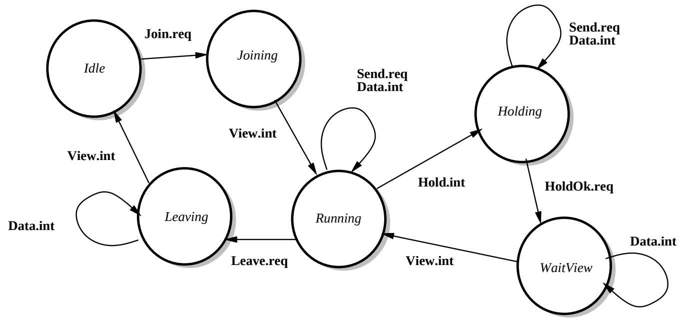


Figure 2: VS Interface State Machine

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 5.

### 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 on to 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 Figure 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, established a new mapping to the HWG specified. This last primitive is offered to minimize the number of accesses to the name service.

Note that, for availability, the name service may be replicated. A possible implemen-

| 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

tation 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 on to 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

```
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 ⟨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 ⟨FLUSH_OK, lwgId⟩ received do
    lwg.nacks := lwg.nacks + 1;
od

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

when ⟨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

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. When the correspondent `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 that 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 messages disseminates the identity of the HWG that should be used during the next view. Thus, the flush protocol is used both to change the group membership and to execute the switch protocol.

## 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. To minimize accesses to the name service, the joining process proposes a mapping based on its own local HWGs, according to the heuristic presented in Section 3. Then, in a single access to the name service it commits this mapping or, in the case where the LWG is already mapped to 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.

```
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 ⟨JOIN, lwgId, processId⟩ received do
    lwg.joiningList := lwg.joiningList ∪ processId;
    lwg.doFlush := TRUE;
od
```

Figure 4: The Create/Join Protocol

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 the `joiningList` and `doFlush` flag is activated. 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. 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.Leave.req ( lwgId, processId ) do
    lwg.state := Leaving;
    hwg.Send.req ( lwg.CurrentHwg, ⟨LEAVE, lwgId, processId⟩ );
od

when ⟨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, that 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

Although this paper is not concerned with describing the policies that trigger the switch procedures, for self-containment, the switch protocol is briefly presented. 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 a ⟨OPEN, lwgId, hwgTo⟩ message on `hwgFrom`. 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.

When a member of the LWG detects that all members have joined *hwgTo*, it sets the variable `nextHwg` and activates the `doFlush` flag. As in the previous cases, this will trigger the execution of the flush protocol that 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 ⟨OPEN, lwgId, hwgTo⟩ received through hwgFrom do
         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 failure handling protocol is quite simple because all 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 on that HWG (see Figure 7). The Light-Weight Group Service waits for an acknowledgment from every LWG 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 .

### 4.9 Synchronization with the Name Server

When a switch occurs, the name service is informed of the new mapping such 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 about the new

```
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

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. The complete protocols are not presented here due to lack of space.

## 5 An Implementation in Horus

### 5.1 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 small 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.

## 5.2 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 this paper. However, the implementation of the flush protocols in Horus, both in the MBRSHIP layer and in the LWG layer, uses a coordinator based approach to reduce the number of multicast messages exchanged.

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 in a coordinator based version where the FLUSH_OK and VIEW messages carry the causal dependencies required to automatically flush data messages.

### 5.3 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 transport protocol we used is UDP/IP with the Deering multicast extension. Each machine runs only one process in our test.

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. To evaluate the effectiveness of the LWG layer, the exactly same tests were run on the traditional HWGs also. For the LWG test, every group member has the stack "LWG:MBRSHIP:FRAG:-NAK:COM" underneath it. Whereas for the HWG test, every member executes on top of the stack "MBRSHIP:FRAG:NAK:COM". 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 fails or leaves the group, the flush time is measured between the detection of the problem and the installation of a new view.

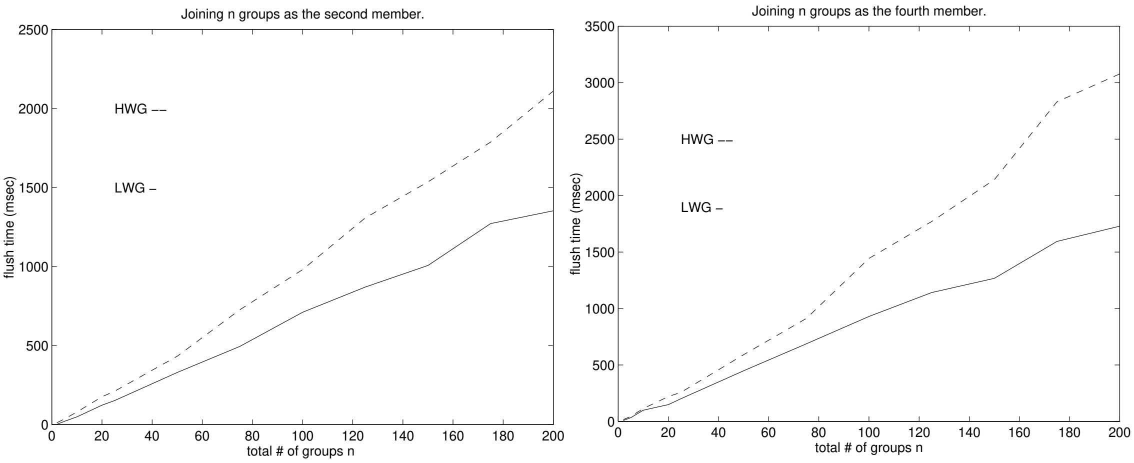


Figure 8: Joining time

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 8 shows the flush time when a process joins as the second and fourth member. In each graph, the flush time for HWGs can be expressed as $F(n) = F_{\mathrm{HWG}} \times n$, where $F_{\mathrm{HWG}}$ is the amount of time for each HWG flush, and $n$ is the total number of groups. The flush time for LWGs can be expressed as $G(n) = F_{\mathrm{HWG}} + F_{\mathrm{LWG}} \times n$ where $F_{\mathrm{LWG}}$ is the amount of time for each LWG flush. When the process joins the first of the $n$ LWG groups, it has to join the underlying HWG first, as a result, the first join involves a HWG flush and a LWG flush. From Figure 8 we can see that $F_{\mathrm{HWG}} > F_{\mathrm{LWG}}$. The reason is described below.

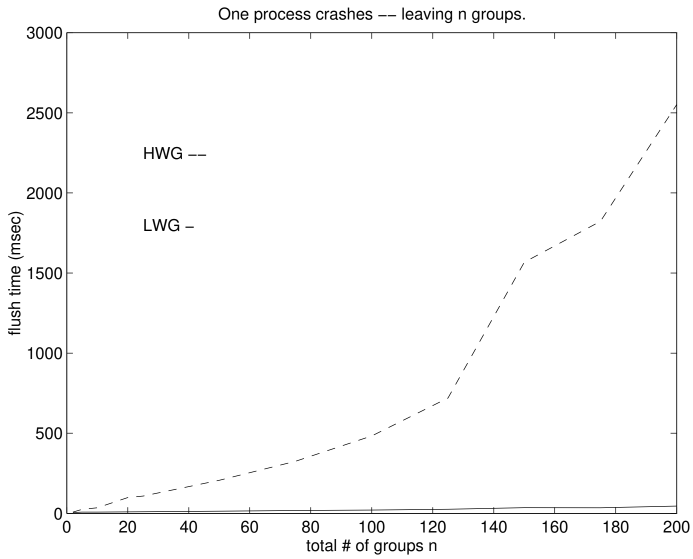


Figure 9: Recovery from crashes (comparative)

In the Horus implementation, the flush process is identical for both HWGs and LWG. In either case, the coordinator waits until it has collected FLUSH_OK messages from all other members. After a flush is done, a new view is installed. In order to install a view in a HWG, a member needs to install the same view in all the underlying layers: MBRSHIP, FRAG, NAK and COM, only after it gets confirmation from all the layers, can a HWG member deliver the `View.int` to its application. On the other hand, when a LWG member installs a view, it can directly send its its application the `View.int`, since the view of the HWG cannot change during a LWG flush. The difference between $F_{\mathrm{HWG}}$ and $F_{\mathrm{LWG}}$ solely comes from the installation of the view in lower layers in HWG, so is the difference between $F_{\mathrm{HWG}}$ and $F_{\mathrm{LWG}}$ (it is observed that these parameters are constant, with $F_{\mathrm{HWG}}$ being 3 milliseconds higher than $F_{\mathrm{LWG}}$).

To evaluate the effect of LWGs on failure recovery, we conducted two tests. In both tests, there are $n$ identical four-member groups.

Figure 9 shows the flush time when one process crashes. Since a failure is notified at each of the $n$ groups, each group starts its own flush. The total flush time for HWGs shows a more than linear increase as $n$ increases, whereas for LWGs , the total flush time increases linearly with a very flat slop.

In the HWG test, there are $n$ HWG flushes running in parallel, whereas in the LWG test, there are $n$ LWG flushes and one HWG flush running in parallel. If the processors in our system have infinite processing power and the network has infinite bandwidth, the flush time for both tests should be equal to the flush time for one HWG flush. But this is not true in reality.

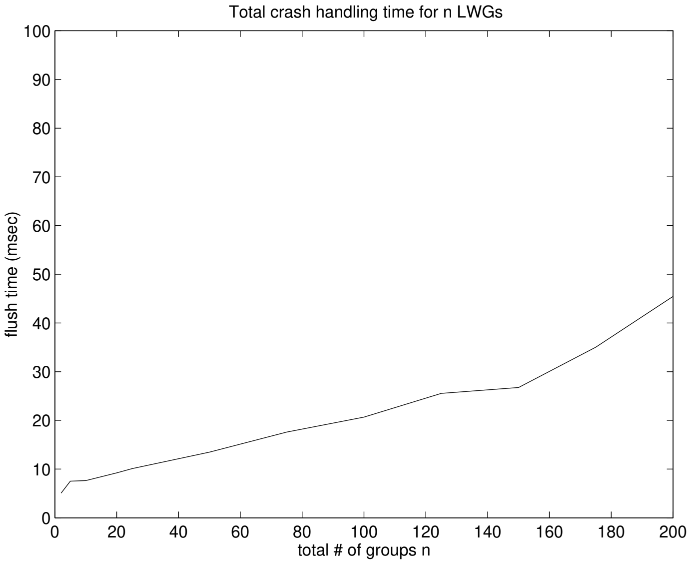


Figure 10: Recovery from crashes (LWG )

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 one "status" report every 2 seconds. When there are $n$ HWGs on each process, a total of $n/2$ 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 total time for $n$ HWG flushes increase dramatically. Another contributing factor to the total flush time is that since $n$ HWG flushes are in parallel, the coordinator process must handle $(4-1)n = 3n$ FLUSH_OK replies during short period of time, this becomes an implosion problem when $n$ is large.

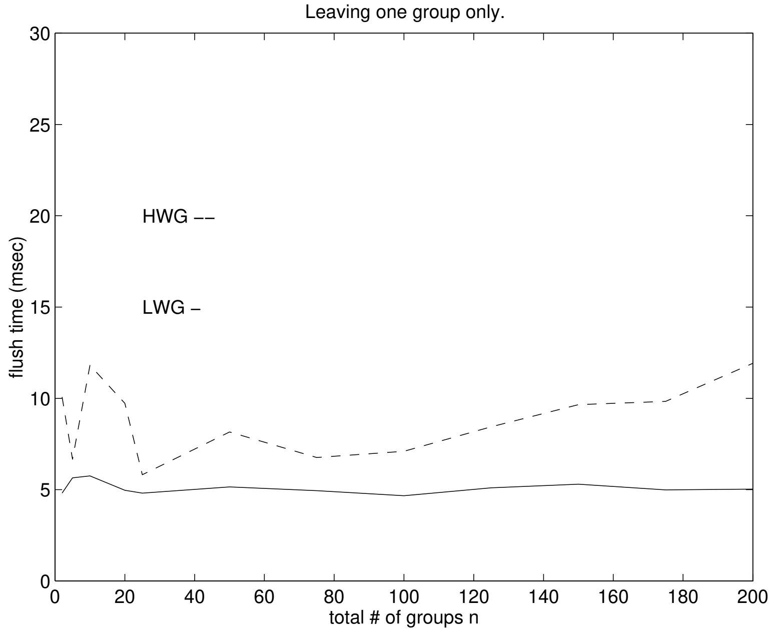


Figure 11: Group leaves

In the LWG test, there is only one HWG for all the $n$ LWGs. Every 2 seconds, each process multicasts one "status" report message in the HWG, therefore receives only one "status" report. The background messages in this case has very limited effect on the system load. Since there is only one HWG flush going on, the coordinator process only needs to collect 3 FLUSH_OK replies. The linear increase in the total flush time as depicted in Figure 10 comes from the fact that there is only one coordinator process delivering FLUSH_OK events to $n$ LWG applications.

Figure 11 shows the flush time measured when a non-coordinator member leaves one of the $n$ groups. This invokes flush in one group only. For HWGs, $F(n) = F_{\mathrm{HWG}}$ and for LWGs, $G(n) = F_{\mathrm{LWG}}$.

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. Figure 12 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. This is because in the HWG test, there

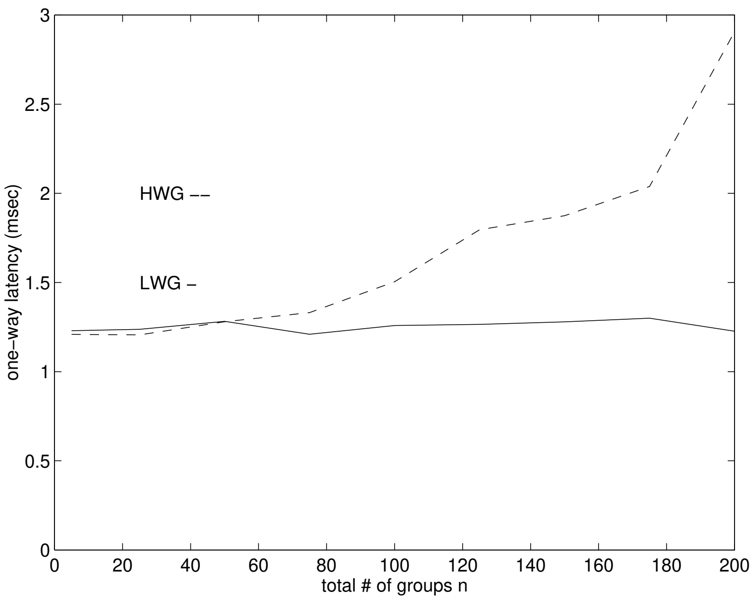


Figure 12: Data transfer

are $n/2$ background "status" report messages arriving at each process per second, whereas in the LWG test, there is only one arriving report message every 2 seconds. The linearly increasing number of report messages in the HWG test contributes to the more than linear increase in one-way latency.

## 6 Conclusions and Future Work

In this paper we have presented a technique that promotes resource sharing among user 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 with switching heuristics (that dynamically modify these mappings) 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. Technical report, Department of Computer Science, Cornell University, March 1995.

[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. Technical report, Cornell University, 1996. (submitted for publication).

[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] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, Fakultat der Universitat Zurich, 1995.

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

[12] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Grade, P. Veríssimo, and K. Birman. A dynamic light weight group service. Technical report, INESC/Cornell Univ., March 1996.

[13] 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. IEEE.

[14] 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.

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