Virtual Synchrony Semantics: A Client - Server Implementation

by

Igor Tarashchanskiy

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2000

Certified by .....
Idit Keidar
Postdoctoral Research Associate
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Virtual Synchrony Semantics:

A Client - Server Implementation

by

Igor Tarashchanskiy

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Group communication systems (GCSs) with virtual synchrony (VS) semantics have proven to be powerful abstractions for distributed fault-tolerant application development. In this thesis, we present an implementation of a virtally synchronous service which is a part of the emerging *Xpand* GCS. This service is based on a new fast VS algorithm. This algorithm is a part of the architecture that separates the multicast and the membership services in order to parallelize their operations. The performance measurements of the system have shown that the commuication overhead is low and the parallelism of the membeship and VS services is effective.

Thesis Supervisor: Idit Keidar
Title: Postdoctoral Research Associate

Acknowledgments

I would like to thank Idit Keidar, my advisor, and Roger Khazan for providing valuable insights and suggestions. I also would like to thank Tal Anker, Ilya Shnaiderman and Gregory Chockler for sharing with me their experience in computer system development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Group communication is a means for providing multi-point to multi-point communication among a group of processes. A group communication abstraction provides application writers with reliable multicast communication services within dynamically changing groups. It also provides membership services which inform group members when other members crash, leave, or join the group.

A group communication system (GCS) allows clients to join one or several *multicast groups* and informs clients about membership changes via *views*. A view contains a set of currently connected clients in a given group. The views are delivered to a client within the stream of messages multicast by clients to the group. A membership service, which is a part of the GCS, generates views in response to processes joining, leaving or disconnecting from the group. The task of this service is to maintain and distribute the information about the current group membership.

The ability to share common information in a fault-tolerant manner is what many applications require. GCSs are designed to meet exactly this requirement; therefore, fault-tolerance is one of the key properties of any GCS. As a matter of fact, a great deal of distributed fault-tolerant applications already rely on the group communication abstraction because of its useful semantics (see [4, 5, 11]).

A group communication system typically provides a handful of properties [13]. The GCS *safety* properties guarantee that the system never violates certain correct characteristics.

Some of them specify the behavior of the GCS membership service. That is, they state the requirements about the membership views provided by the service. The GCS must also provide certain multicast characteristics regarding the reliability and ordering of messages. An important safety property is *Virtual Synchrony* [13]. It guarantees synchronous message delivery with respect to views. Specifically, if two clients transition from V' to V together, they must deliver the same set of messages in V'.

In addition to the safety properties, a GCS must provide some *liveness* guarantees. They characterize how well a system progresses in delivering messages and views. Liveness of the system depends heavily on the stability of the underlying physical network. However, this is barely a problem because, at least temporary, network stability is a sensible condition in practice. The term virtual synchrony semantics (VS) refers to a combination of these liveness and safety properties that includes the Virtual Synchrony property.

A serious issue with which a virtually synchronous GCS has to cope is maintaining an acceptable level of performance while the number of communicating clients, and the size of the network grow. To implement VS, a GCS needs to perform the synchronization rounds in which the clients exchange the information about their states. As the network latency increases, the synchronization rounds become very costly. Message length and complexity increase with the number of clients.

This thesis implementation is a building block of the emerging **Xpand** GCS designed to resolve many performance issues. Specifically, we implement the VS algorithm by Keidar and Khazan [10], as a part of the architecture proposed by Anker et al. [2]. The novelty of this architecture is the separation of the membership service from the multicast client in order to parallelize the executions of the membership and the VS protocols. Moreover, the VS protocol requires only one round of synchronization messages per each membership change. It uses the external membership service of Keidar et al. [10]. The new features allow the system to operate efficiently on a Wide Area Network (WAN).

The **Xpand** system consists of three main components:

- *Core* implements reliable FIFO multicast [3] among clients.
- The membership server provides clients with views via a TCP interface [9].

- The VS library, called *VS-wsession*, provides the VS service implemented in this project.

VS-wsession supports two types of semantics: *strong* and *weak*. The first one refers to the VS semantics described earlier while the second type of semantics implements only a subset of the VS properties that does not include Virtual Synchrony. The capability to support two semantics makes the entire system more flexible and more useful for applications requiring different semantics. For example, an application can participate in a video conference and edit a shared file at the same time. Clearly, the file consistency requires stronger reliability guarantees than the video.

VS-wsession is a C++ library, which can be linked with the user application. It is implemented using approximately 9000 lines of code. VS-wsession allows the application to establish new or join existing groups in order to multicast and receive messages. The multicast send and receive operations are non-blocking; they return an error in case the lower level operations would block.

VS-wsession also informs the application about new membership views because a typical application needs to know the other parties with which it communicates. The membership views go through VS-wsession from the membership server to the application. However, before VS-wsession can deliver a view to the application, it has to complete one or more synchronization rounds. To parallelize the generation of a new membership view with the synchronization rounds, the membership server notifies its clients, VS-wsessions, when it begins to engage in a membership change in a given group. Thus, the clients can start synchronizing even before they receive the view from the membership server. Once the view is received from the membership server and the synchronization are rounds complete, the view can be delivered to the application.

In this thesis, we present measurements of VS-wsession's performance. The results of our measurements show that the computation overhead associated with message delivery while no membership changes occur is very low. It also shows that the synchronization rounds complete faster than the membership view generation in majority of cases, and the difference in time is approximately 50%.

The following sections describe the system in greater detail. Chapter 2 contains the

specifications of VS-wsession. Chapter 3 describes how VS-wsession interacts with Core and the membership server. Chapter 4 describes the API. The details of the implementation of VS-wsession follow in Chapter 5. This thesis also shows the performance characteristics of the system in Chapter 6.

Chapter 2

Virtual Synchrony Specifications

We now present the specifications of the entire system as conveyed by VS-wsession to its application. Here, a process corresponds to an instance of VS-wsession on a network. Each process is responsible for serving its user application. The communication between a process and its user is defined in terms of a set of events. The send event refers to submitting a messages by the application to the process for multicasting. Similarly, a receive event refers to the delivery of a message by the process, possibly from another process, to the application. The stream of messages delivered to the application contains views. The view consists of a set of communicating processes in a given group and an identifier from an ordered set. The *installation* of a view is the delivery of a message containing the view and a transitional set to the application, as explained below.

Of the specification properties below, VS-wsession simply preserves most of them. The only properties it implements itself are (1.3),(2.3) - (2.5) and (4.1.b). The properties it preserves are guaranteed by the membership service and Core. The details of the interaction of VS-wsession with the two components follow in Chapter 3.

VS-wsession meets the following specifications described in [13, 10]:

2.1 Membership Safety

(1.1) Self Inclusion: If process p installs view V, then p is a member of V.

(1.2) Local Monotonicity: If a process p installs view V after installing view V', then the identifier of V is greater than that of of V'.

(1.3) Initial View Event: Every send and receive event occurs in some view.

2.2 Multicast Safety

(2.1) Delivery Integrity: For every receive event, there is a preceding send event for that message.

(2.2) No Duplication: Two different receive events with the same message cannot occur at the same process.

(2.3) Sending View Delivery If a process p receives message m in view V, and some process q (possibly $p = q$) sends m in view V', then $V = V'$.

(2.4) Virtual Synchrony If processes p and q install the same new view V in the same previous view V', then any message received by p in V' is also received by q in V'.

(2.5) Transitional Set :

1. If process p installs a view V in (previous) view V', then the transitional set for view V at process p is a subset of the intersection between the member sets of V and V'.
2. If two processes p and q install the same view, then q is included in p's transitional set for this view if and only if p's previous view was also identical to q's previous view.

2.3 Ordering and Reliability

(3.1) FIFO Delivery: If a process p sends two messages, then these messages are received in the order in which they were sent at every process that receives both.

(3.2) Reliable FIFO: If process p sends message m before message m' in the same view V, then any process q that receives m' receives m as well.

2.4 Liveness

Before specifying the liveness properties, we need to define, as in [13], a *stable component* and an *eventually perfect failure detector* used by the VS service. The stable component is a group of communicating processes that are eventually alive and connected to each other, but disconnected from all processes not in the group. A failure detector is a module that provides information about liveness of processes on the network. An eventually perfect failure detector is a failure detector that eventually reports the correct information about the processes' connectivity. It is referred as $\diamond P$ in [6, 13]. For more more formal definitions, please see [13].

Liveness is only required if eventually there exists a stable component S in the network (processes eventually stop crashing or recovering), and the failure detector *behaves like* $\diamond P$ [6, 13].

(4.1) Liveness: Assuming that S exists and the failure detector behaves like $\diamond P$, then for every stable component S there exists a view V with S as its members set such that the following three properties hold for each process p in S:

a. **Membership Precision:** p installs view V as its last view.

b. **Multicast Liveness:** Every message p sent in V is received by every process in S.

c. **Self Delivery:** p delivers every message p sent in *any* view unless p crashed after sending it.

2.5 The Utility of Virtual Synchrony and Transitional Set

Virtual Synchrony is a very useful property for many applications that maintain replicated data. The application processes that maintain data replicas join multicast groups through a GCS with virtual synchrony semantics. The operations on the the replicated data are multicast in messages by each process to the entire group. These messages are received reliably and in a FIFO manner. If a process applies the operations indicated in the received messages, the Virtual Synchrony property guarantees that the processes remaining in a group will receive the same sequences of messages and perform the same operations on their data replicas. This implies data consistency among the group members.

Transitional Set allows processes to exploit Virtual Synchrony. Specifically, the transitional set tells an application process which other processes move together with it from their mutual current view into their mutual new view. This way, the process knows which other data replicas remain consistent with itself. For concrete applications of Virtual Synchrony please see [7, 11, 1].

Chapter 3

Environment

The distinction of the new design is that it separates the membership service from the multicast communication. Since the membership changes are relatively infrequent compared to the communication traffic, the new design removes the unnecessary overhead from the major part of the communication traffic and thus provides for better efficiency. This feature makes the system especially suitable for such wide area networks as the Internet. Moreover, the client-server architecture, which is shown in Figure 3-1, improves the scalability of the system.

3.1 Membership Service

A set of servers communicating among themselves and with their VS clients via TCP sockets implement the membership service [9]. Each server uses its TCP interface to communicate two types of messages to its clients:

1. **Start-Change Message:** A start-change message containing an identifier tells a client that the membership is changing. It provides an approximation view of the membership of the next view. It also carries an identifier.

2. **New-View Message:** A new-view message contains a set of clients that the membership servers have determined to be in the next view. The new-view message also

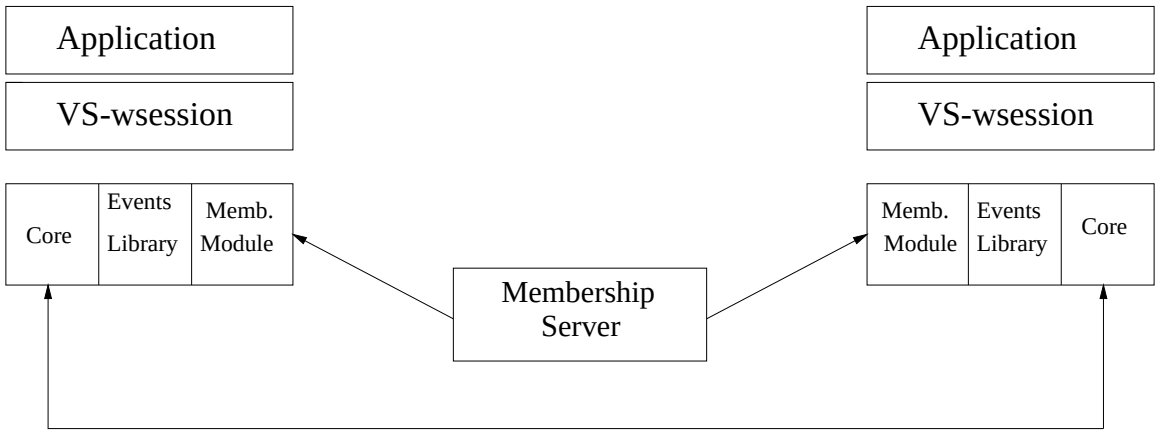

Figure 3-1: VS architecture

contains a view identifier and the last start-change identifiers received by each group member.

These messages are provided to client's VS-wsession. The identifiers of the start-change and the new-view messages are guaranteed to monotonically increase. Moreover, the membership of the new-view message is guaranteed to be a subset of set given in the last start-change message. They also guarantee properties (1.1), (1.2) and (4.1.a).

Each VS-wsession uses the *membership interface module* provided by **Xpand** in order to retrieve the messages from its membership server. The information available in the start-change allows to a client to start the synchronization round with the set of clients indicated in the message. The new-view allows each client to determine the transitional set and to agree with other clients on which message to deliver before installing the message's view.

3.2 Core as Reliable FIFO Layer

The VS-wsession is linked with Core, which provides reliable FIFO multicast channels on top of UDP/IP-multicast [3]. Core maintains a *connected set*. This set is a set of clients with whom Core maintains reliable FIFO communication. Core buffers all received messages until it is told to deallocate them.

Core's interface provides the following operations:

1. Multicast messages to a given communication group;

2. Receive messages delivered to a given group from clients;

3. Re-multicast or re-unicast to a specific client messages delivered from other clients, VS-wsession uses this operations to forward messages;

4. Change the connected set;

5. Inform Core when a particular message can be deallocated.

The protocol implemented by Core guarantees properties (2.1), (2.2), (3.1), (3.2), and (4.1.b-c).

The UDP protocol has been chosen for Core over TCP for performance reasons. Even though the TCP protocol guarantees all the required properties, it can only support point-to-point communication. On the other hand, the UDP protocol can multicast messages via IP-multicast. The use of multicast eliminates the need to send the same message multiple times and thus alleviates the network load.

Core compensates for UDP's unreliability and reordering. It provides FIFO message ordering between any pair of clients in a group. Core uses an ACK/NACK algorithm with timeouts. It buffers all in-bound and out-bound messages, acknowledges received messages, sends negative acknowledgments in case it detects missing messages, and retransmits a message after receiving a NACK for it.

Core allows VS-wsession to use its message buffer space to avoid multiple message copying. When VS-wsession informs Core about a message deallocation, Core will "garbage-collect" all the messages up to and including the indicated one according to the message sequence numbers. A more detailed discussion of Core can be found in [3].

3.3 Support for Multiple Groups

The VS-wsession also uses the membership interface module for group multiplexing [3]. VS-wsession creates and manages strong and weak groups. VS-wsession guarantees virtual synchrony semantics in each strong group. The application can be a member of any number of

groups limited by a configurable constant MAX_GROUP_NUM. The groups are referenced by names, which are character strings with maximum length of MAXGRPNAME. Core resolves the group names into group identifiers and network addresses internally [3] while VS-wsession uses the string representation.

Chapter 4

Virtually Synchronous Client Interface

As mentioned above, the objective of VS-wsession is to provide the application with the virtually synchronous semantics using the membership and Core. As for the application interface, VS-wsession allows the application to join and leave existing groups (known to the client's server). If the application attempts to join a non-existent group, the group will be established with a single member. Once the application has chosen to join one or more strong groups, the VS-wsession joins these groups at the membership server and starts delivering application messages in a manner that is synchronous with respect to the membership events.

VS-wsession also generates *block* events with special blocking messages injected into the application message stream. A block event notifies the application about a pending view installation. The application must finish sending its messages in the current view and subsequently respond with a *block-OK* message generating a block-OK event at VS-wsession. The block-OK event allows VS-wsession to run the synchronization round for the pending view installation. The application will be unblocked when VS-wsession delivers the new view [10].

The current implementation of the VS client is single-threaded; it relies on **Xpand**'s *events* library. The thread of control is shared among the user application, the VS client, the membership interface module, and Core based on demand and priority. Every module registers its event handlers as *call-back* routines with the events library, which is responsible

for scheduling these call-backs.

In order to receive information from VS-wsession, the application must provide delivery and unblock call-backs upon join. These routines will be called to deliver a message or an unblock event. In case of the weak service group, the user also needs to supply a membership change call-back. For the strong groups, the third call-back is not needed because the VS-wsession will notify the user about a new view through the delivery call-back using a special new-view message.

VS-wsession uses the following messages to communicate with its application:

1. **Application Message, APP_MSG:** This message is the application data treated as a byte buffer sent and delivered among group members.

2. **New View Message, NVIEW_MSG:** The new view message contains a membership and a view identifier. It also contains a transitional set. After receiving this message, the application can continue multicasting messages in the new view.

3. **Block Message, BLOCK_MSG:** The block message is used to notify the application about the start of a view reconfiguration.

4. **Block-OK Message, BOK_MSG** Upon receiving the block message, the user application must send any relevant messages in the current view and terminate them with a block-OK message. The block-OK message tells the client that the application will not send any more messages in the current view.

The VS-wsession library provides only per group semantics and can support the weak groups as well as strong ones. It has the following user API.

4.1 VS-wsession User API

W_Init()

`arguments:`	`char string`	*client_name*
	`char string`	*server_address*
	`unsigned short`	*server_port*

membership type *type*

checks: if *type* $\in$ {WEAK, STRONG}.

effects: instantiates and initializes the VS-wsession abstraction
for a new client;
if *type* = STRONG, VS-wsession will allow to create
only STRONG groups.

returns: a new wsession handle.

W_Close()

arguments: wsession handle *weph*

checks:

effects: disconnects from the membership server and deallocates all
the data structures for this client.

returns:

W_Join()

arguments: wsession handle *weph*
char string *group_name*
function pointer *unblock_callback*
generic pointer *unblock_parameter*
function pointer *delivery_callback*
generic pointer *delivery_parameter*
function pointer *membership_callback* (NULL for STRONG (VS) groups)
generic pointer *membership_parameter* (NULL for STRONG (VS) groups)
membership type *type*

checks: if this client is already a member of the requested group;
if *type* $\in$ {WEAK,STRONG};
if *type* $\neq$ WEAK in case of the STRONG wsession type.

effects: instantiates and initializes a new group object;
issues a join event for the appropriate group to
the membership server;
notifies Core.

returns: new group handle.

W_Leave()

arguments: wsession handle *weph*
group handle *group_ID*
checks:
effects: issues a leave event to the membership server;
notifies Core;
deallocates the group instance and removes call-backs.
returns:

W_Send()

arguments: wsession handle *weph*
group handle *group_ID*
unsigned short *buffer_length*
char buffer *buffer*
checks: if *buffer_length* $\leq$ MAX_VS_USER_MSG_SIZE.
effects: sends an APP_MSG containing *buffer*
to the group with *group_ID*.
returns: 0 if message is sent;
-1 if message cannot be sent presently.

W_Group_Poll()

arguments: wsession handle *weph*
group handle *group_ID*
checks:
effects: polls Core for available messages to receive in
the group specified by *group_ID*.

returns: number of deliverable messages.

W_Group_Receive()

arguments: wsession handle *weph*
message pointer *msg_pointer*
group handle *group_ID*

checks:

effects: performs non-blocking receive;
if there is a message to deliver, *msg_pointer* points
to the message;
else *msg_pointer* is set to NULL;

returns: number of bytes in *msg_buffer* if message is APP_MSG;
number of clients if message is NVIEW_MSG;
0 if message is BLOCK_MSG;
-1 if no messages available.

The contents of *msg_pointer* are deallocated once control returns from application to VS-wsession.

W_Copy_Group_Receive()

arguments:	wsession handle	*weph*
	message pointer	*msg_pinter*
	unsigned short	*maximum_length*
	group handle	*group_ID*
	user message info pointer	*info_pointer*

checks:

effects: same as above, except that message contents
are copied to *msg_pointer*;

returns: same as above;

W_Block_Ack()

arguments:	wsession handle	*weph*
	group handle	*group_ID*

checks:

effects: acknowledges BLOCK_MSG and agrees not
to send messages until next view.

returns:

Chapter 5

VS-wsession Implementation

5.1 Roadmap

This Chapter is structured as follows: Section 5.2 describes how VS-wsession works with other **Xpand** modules. Section 5.3 explains the advantages of the event-driven design of VS-wsession. Section 5.5 presents an overview of the VS service algorithm [10]. Section 5.6 describes the main data structures of VS-wsession and how they are used by the algorithm. Section 5.8 elaborates on the forwarding strategy employed by the algorithm [10]. Section 5.9 explains the stability tracking mechanism of VS-wsession. Finally, Section 5.10 exposes the limitations of VS-wsession.

5.2 Interconnection with Other Modules

The call-backs provided by the application to VS-wsession for a weak group are registered directly with Core and the membership interface module. For the strong groups, the two application call-backs are registered with the VS-wsession instead.

In order to receive messages from the network, VS-wsession registers its own delivery and unblock call-backs with Core. In addition, VS-wsession registers a call-backs with the membership interface module to handle the messages arriving from the membership server. All the call-backs are invoked in a context of a group.

5.3 Event-Driven Design

A single-threaded control managed by the events library has been preferred because the entire system is inherently event-driven. Namely, all computations are triggered either by network events, or by the user application.

Another natural reason to adhere to the single-threaded event-driven paradigm is the complexity and the performance overhead associated with data synchronization. With data heavily shared among all groups on the level of the Core layer, it is not clear how to distribute the work among multiple threads.

Although less significant, portability is still another reason to avoid multiple threads. Different platforms are optimized for different thread packages. Even though POSIX seems to be standard for the Unix platforms, Microsoft Windows systems perform noticeably better with their native threads libraries.

5.4 Communication Messages

The client manipulates a set of messages to communicate with the membership server, other VS clients in its groups, and the user application. The communication with the user applications, which has been described in Chapter 4, is local. That is, the application is linked with an instance of VS-wsession. The descriptions and formats of the remaining messages are presented below.

Membership server to VS client communication:

1. **Start-Change Message:** The start-change message is sent by the membership server to its clients whenever the server engages in a membership change. This message provides an approximation of the new membership. Each start-change message also contains an identifier, which increases monotonically with respect to each server. The purpose of this identifier will be mentioned during the description of the synchronization message.

2. **Membership View Message:** The membership view message from the server contains a set of clients that the membership servers have determined to be in the next view. This set of clients is guaranteed to be a subset of the view provided in the preceding start-change message. The membership view message also contains a view identifier and a mapping from view members to the last start-change identifiers received by each of them. The monotonicity of the view identifiers is also guaranteed by the server.

Client to client communication:

1. **Application Message:** This is the message sent and received by application as described in Chapter 4. The application message must not exceed MAX_VS_USER_SIZE bytes in order to be successfully multicast. It is the responsibility of the application to make sure that the messages it tries to communicate via the VS-wsession do not exceed this limit. The application messages carry sequence numbers when sent among clients.

2. **Synchronization Message, also Cut, CUT_MSG:** VS-wsession multicasts a cut message every time it receives a start-change message from the server. The cut includes the current view, including the start-change mapping, and the view identifier. It also contains the sequence numbers of the last messages it can potentially deliver to the application from each view member at the moment of sending this cut. These are the messages received from Core. It also has a cut identifier, which is equal to the start-change identifier in the start-change message received before the cut.

3. **Stability Message, also Stable Cut, STBL_MSG:** VS-wsession uses the stable cut message to communicate the message stability information currently available to it to other VS-wsesion instances. The stable cut specifies the last message delivered to the application from each sender in the group.

4. **View-Start Message, also VSTRT_MSG:** The view-start message is identical to the membership view message in content, but is multicast within the group rather than

sent from the server to its clients. It identifies the end of the application message stream in the old view and notifies other group members about this member's transition to the new view.

5.5 The Algorithm Overview

While the network is stable and the client has installed at least one view in the group, VS-wsession immediately delivers all the messages from Core to the application through the delivery call-back registered by the user when it joins. When a view reconfiguration occurs, the VS algorithm is triggered.

When the membership server informs VS-wsession (via the appropriate call-back) about a pending view with a start-change message, VS-wsesion blocks the user application from sending further messages. Then, after the application replies with a block-OK message, VS-wsession multicasts a synchronization message tagged with the identifier provided in the last start-change message. That is, the identifier of the synchronization message is equal to the identifier of the start-change message.

To keep track of the application messages received from Core, VS-wsession assigns VS sequence numbers to them; these sequence numbers are different from those used by Core. The synchronization message describes which application messages VS-wsession is capable of delivering by specifying the VS sequence number of the last message received from Core from each member. In order to guarantee Property (2.3), all the VS-wsessions in the group have to agree on the set of messages they deliver to the application. After collecting all the synchronization messages and receiving the view from the membership server, each instance of VS-wsession can determine the appropriate set, as explained below.

Because VS clients operate completely asynchronously and because multiple membership changes may occur, VS-wsession can receive multiple start-change and synchronization messages before the membership view arrives. The start-change identifiers mapping included in the membership view tells VS-wsession which synchronization messages from other members to consider. Using the information contained in these synchronization messages, this VS-wsession decides on the set of application messages it must deliver in its current view. Since

the same view sent to all VS-wsessions contains the same start-change identifiers mapping, VS-wsessions use the same synchronization messages to agree on the correct set of messages.

Once VS-wsession receives a new view V from the membership server, it can determine the intersection of the current view V' and V. After VS-wsession collects a synchronization message whose identifier is equal to the respective start-change identifier given in V from each member of the intersection, it computes the transitional set of views V and V' to include every member of the intersection whose synchronization messages corresponding to V contain the same view V'.

VS-wsession delivers the maximum number of messages with respect to the transitional set. That is, for each sender, VS-wsession delivers the maximum number of messages indicated in the cut messages of the members of the transitional set.

VS-wsession may not have all the messages it has to deliver before transitioning to the new view because some members of the transitional set may have delivered more messages from already disconnected clients than this VS-wsession. In this case, according to the forwarding strategy explained below, some members of the transitional set must forward the necessary messages to this VS-wsession. This way, the local VS-wsession can satisfy the VS conditions and install a new view.

This algorithm operates in parallel with the membership service algorithm and requires a single round of synchronization messages per every start-change event. Assuming the low frequency of the membership changes, the algorithm minimizes the reconfiguration period during which the application is blocked from sending and makes the high latencies of the wide area networks tolerable.

5.6 Data Structures and Algorithmic Details

This section describes how VS-wsession data structures are used by the algorithm. Most of the VS-wsession data structures implement the VS state described in [10].

- **WEPID:** Every client in a group has a *WAN end-point* identification containing client's name, membership server IP, client's IP, UDP port number and some other information

used by Core. Thus, the clients are referenced by their WEPIDs in all local data structures.

- **Reliable Set:** The VS-wsession maintains a reliable set, the set of clients to whom Core has to maintain reliable communication. This reliable set is equal to the currently installed membership view during normal operation. Once a start-change message is received, the reliable-set becomes the union of the start-change set and the current reliable set. Ultimately, when a new membership view is installed, the reliable set is reset back to normal, that is, to the view's membership.

- **Message Data Structures:** In addition, VS-wsession needs to keep track of the current view, the last start-change message from the membership server, and the last synchronization messages from all other VS-wsession instances. The VS-wsession stores only the last start-change and corresponding synchronization messages because the system does not deliver obsolete views. (This is different from the algorithm description in [10]). In other words, if this VS-wsession receives a new start-change message, it will deliver a view no older than the one corresponding to this message – all the intermediate views will be skipped. Similarly, if this VS-wsession receives a new synchronization message from another group member, it will deliver a view no older than the one corresponding to this synchronization message, which in turn corresponds to the start-change received by the other member. As mentioned in Chapter 5.4, the correspondence between the start-change and synchronization messages is established through their identifiers.

  Analogously to the synchronization messages, VS-wsession stores the last view-start message from each. Since the VS-wsession instances operate asynchronously with different speeds, each instance needs to know the view of each other one in order to guarantee property (2.3). Thus, each message received from Core is associated with the correct view.

- **Send and Receive queues:** VS-wsession also maintains a send queue and a receive queue. The receive queue is cleared after the application delivery call-back is invoked

forcing the application to dequeue every message delivered to it. The send queue is not used for application messages, but only for internal messages such as cuts, stable cuts, and view-start messages. These are put on the queue until the Core is ready to multicast them. VS-wsession tries to flush the send queue before every send operation.

- **Application and Forward Set Message Buffers:** The most complex data structures of VS-wsession are the message buffers. Both the application message and the forward set buffers are multi-dimensional data structures that are indexed by a view, a message source WEPID, and a message VS sequence number.

  While in normal mode, that is, no membership events are pending, the messages from the network are put on the receive queue and delivered to the application. However, before invoking the application delivery call-back, VS-wsession saves each message in the forward set message buffer. The forward set contains all the messages that may have to be forwarded to other members during the next view change. The messages in this buffer are "garbage collected" either by the stability tracking mechanism after they have been universally delivered, or during view reconfigurations along with all other data structures.

  If a view change is taking place, VS-wsession will first verify if the application message is delivery-safe according to the VS semantics. A message is is delivery-safe if: VS-wsession has no current start-change message, or VS-wsession has not send a synchronization message corresponding to the current start-change, or VS-wsession contains a synchronization message from a member of the current transitional set and which has committed to deliver this message. If not, the message will be put into the application message buffer for potential later delivery. The client attempts to empty the application message buffer every time it receives a membership view, a synchronization message, or another application message.

  Ultimately, the messages stored in the application message buffer either become safe to deliver before the new view is installed, or they are discarded, and the storage is "garbage collected". The safe messages will be delivered to the application and buffered in the forward set as described above.

5.7 Garbage Collection

Eventually, if the network stabilizes (no membership events are generated),VS-wsession will receive the last membership view message from its membership server. This will allow the VS algorithm to determine the transitional set and the set of messages to be delivered from and forwarded to all other group members. Before delivering the new view to the application and multicasting a new view-start message, VS-wsession reclaims the space of all data pertaining to the old views. This includes all types of messages sent and received in the previous views plus some bookkeeping data. The subsequent new view delivery will also unblock the user application, and the client will resume its normal operation.

5.8 Forwarding Strategy

VS-wsession relies on a forwarding strategy that uses the transitional set information in order to minimize the number of the forwarded messages. Once the transitional set has been determined and all the synchronization messages have been collected, VS-wsession can find out which clients miss which messages. They determine which messages each VS-wsession must forward. If more than one instance of VS-wsession is capable of retransmitting a certain message, the one with the minimum WEPID will retransmit. The missing messages from the members of the transitional set will not be forwarded because these members must still be connected through the Core, which must eventually deliver the messages. Before VS-wsession installs a new membership view, it forwards all the necessary messages in the current view from its forward set buffer, using Core's re-multicast/re-unicast operation. Since the current view may stay unchanged for a long time, in order to avoid forward set buffer overflow, VS-wsession employs stability tracking as explained in the next section.

5.9 Stability Tracking

VS-wsession maintains a *stability matrix* that is a set of stability cut message, one from each member of the current view. While the view is stable (no membership reconfiguration in

progress), the VS-wsession periodically multicasts stability tracking information. Specifically, whenever the local message count exceeds BUFFER_LIMIT, VS-wsession multicasts a stable cut. Upon receiving a stable cut message, VS-wsession can determine if this cut belongs to the current view using the saved view-start message of the source. If so, VS-wsession will add this cut to its stability matrix and will attempt to "garbage collect" application messages relying on the information in the updated matrix as explained below.

If the matrix contains stable cuts from all the members of the current view, VS-wsession determines the minimum entry of all the stable cuts corresponding to a certain sender, and "garbage collects" all the messages in its own forward set buffer up to this minimum. VS-wsession repeats the procedure for all the members of the current view. This type of stability tracking has been chosen because of its simplicity; it can be fine-tuned by adjusting BUFFER_LIMIT, which is the minimum number of buffered messages before the next stable cut is sent. The stability matrix size is $\Theta(\#clients^2)$. For groups of several hundred members, this is a tolerable memory overhead.

5.10 VS-wsession Limitations

The maximum message size is limited by the UDP packet size, 64 Kbytes, because the current implementation is not capable of message fragmentation and reassembly.
MAX_VS_USER_MSG_SIZE (see Section 4.1) is somewhat smaller due to headers. Consequently, the system is currently restricted in two ways. First, the user application must fragment and reassemble its own messages if it wants to send longer messages. Second, the group size must not exceed ≈ 450 members; otherwise, some internal VS messages like cuts would not fit into a single message.

More importantly, the system's scalability is limited as a result of the linear message size growth with respect to the number of members in a group. The sizes of the start-change, new view, view-start, and the synchronization messages are directly proportional to the group size. Thus, the number and the size of the messages required to guarantee VS grow linearly. The scalability can be improved by adding hierarchy to the system as in *Structured Virtual Synchrony* [8], which is explained in Chapter 7.1.

Chapter 6

Performance Measurements

The system performance was measured on a LAN. The goal was to measure two characteristics:

1. The time overhead associated with the delivery of an application message from **Xpand**'s Core to the user application at normal times.

2. The time required to satisfy Virtual Synchrony during a view reconfiguration.

The next section describes the group configuration used for the measurements. Section 6.2 explains the overhead associated with normal message delivery. Section 6.3 describes the overhead of the VS algorithm during a view reconfiguration.

6.1 Measurement Setup

The measurements were conducted on three machines: Pentium III(850 MHz, 512 MB RAM), Pentium II(400 MHz, 128 MB RAM), and Pentium Pro (200 MHz, 256 MB RAM). All three machines ran Red Hat Linux (Version 2.2.14-5), and all three were connected to a LAN whose round-trip times for 8 Kbytes and 500 bytes were 17 msec and 2.5 msec respectively. All three machines did not run any user processes except for a client running VS-wsession, a membership server, and an X-server. The data presented on the figures below were collected on the fastest machine.

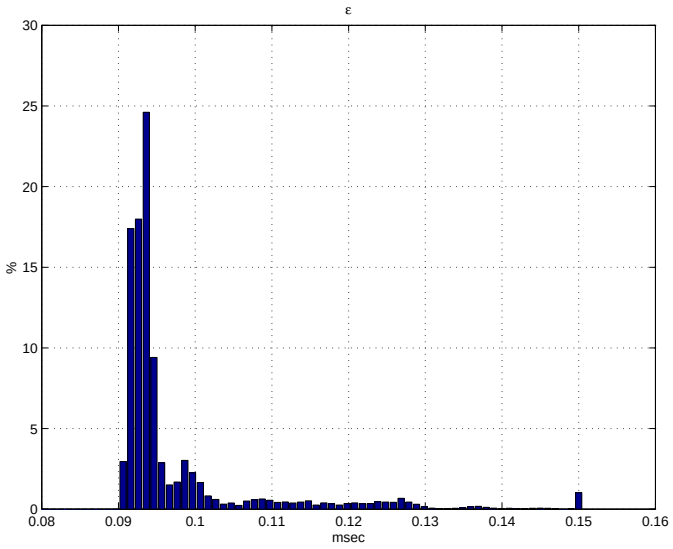

Figure 6-1: Distribution of VS message overhead, ϵ.

6.2 Overhead at Normal Delivery

We denote by ϵ the time since an application message is available from Core until its delivery by VS-wsession to application. Figure 6-1 shows the distribution of ϵ on the fastest machine for the described group configuration of 3 membership servers and 3 VS clients, one client per each server. In this configuration, each client sends an 8 Kbyte message every second. The average and the median of 8900 samples are 98.5 μsec and 94 μsec, respectively on the fastest machine.

During normal operation, that is, when no membership events are generated, the message overhead is well below 1 msec on all machines. On the fastest machine of those three, shown in the figure, the numbers are very close to 100 μsec. The numbers for the other two machines are proportional to their computer speeds:

- avg/median = 221.7/210 μsec for Pentium II;
- avg/median = 419.2/406 μsec for Pentium Pro;

These results show that the message delivery overhead is negligible. It is smaller than the network delay of a message delivery even on a LAN by more than one order of magnitude.

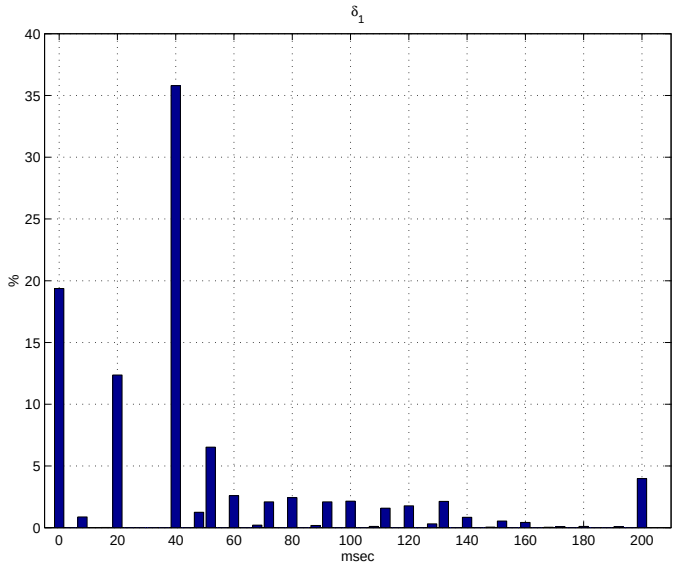

Figure 6-2: Distribution of time to receive a membership view, δ_1.

6.3 Virtual Synchrony Algorithm Duration

The following definitions explain the notation used for the figures in this Section:

- δ_1 = the time since the start-change message until the following new view message;
- δ_2 = the time since the first start-change message until all synchronization messages are collected for the subsequent view;
- δ_3 = the time since the first start-change message until new view is delivered to application, This is equal to $max(\delta_1, \delta_2)$+ computation time.

Figures 6-2 , 6-3 , and 6-4 display the distributions of δ_1, δ_2, and δ_3 respectively. They have been computed for the aforementioned configuration with a fourth client joining and leaving every 5 sec. The number of membership events registered on each machine is 8900. No noticeable difference between join and leave events has been observed. The messages used in the synchronization algorithm are of size $\approx$ 500 bytes. The fourth client that generates the membership events joins and leaves the group via the membership server running on the Pentium II machine.

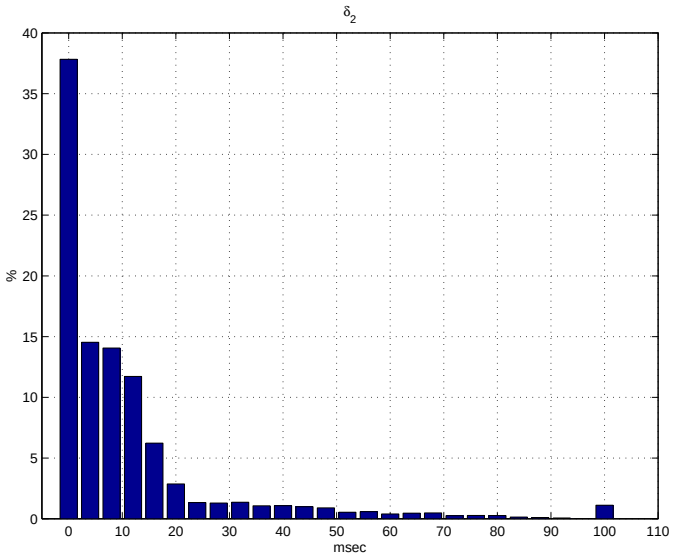

Figure 6-3: Distribution of time to collect all synchronization messages, δ_2.

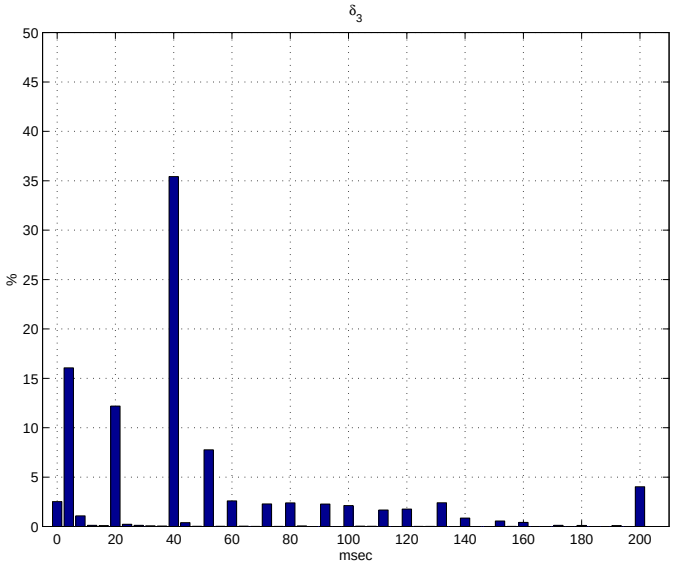

Figure 6-4: Distribution of the total view reconfiguration time, δ_3.

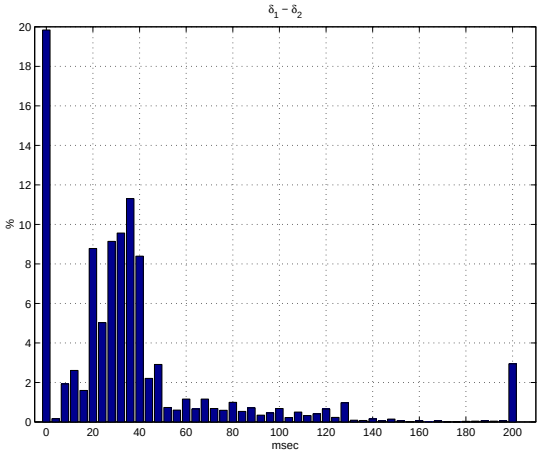

Figure 6-5: Distribution of the difference between δ_1 and δ_2.

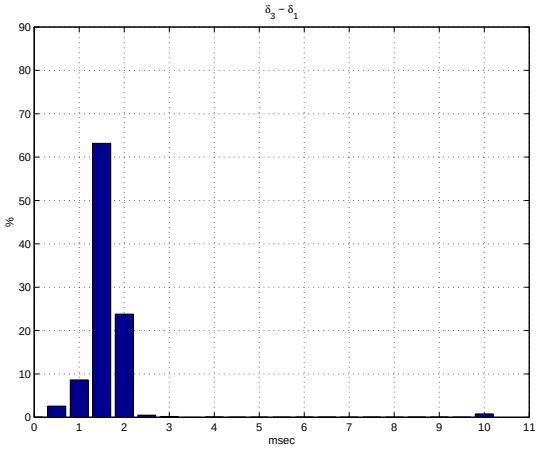

Figure 6-6: Distribution of the difference between δ_3 and δ_1.

Table 6.1: Average and median times for the VS algorithm duration.

		δ_1	δ_2	δ_3	$\delta_1 - \delta_2$	$\delta_3 - \delta_1$
Pentium III	avg (msec)	50	13	52	38	1.9
	median (msec)	40	5	41	30	1.6
Pentium II	avg (msec)	116	67	120	49	4.0
	median (msec)	80	47	84	38	3.4
Pentium Pro	avg (msec)	76	30	83	48	7.2
	median (msec)	50	10	56	37	6.6

Figure 6-5 shows the distribution of the difference between δ_1 and δ_2. The difference is consistently positive and $\approx 50\%$ of δ_1 in magnitude. This indicates that the VS synchronization completes by 30 msec on average faster than the membership algorithm.

Since δ_1 is consistently greater than δ_2, Figure 6-6 shows the computation overhead of the VS algorithm after all the synchronization and the membership messages are received.

The averages and the medians of the six quantities on all three computers are presented in Table 6.1.

The distributions of δ_1 and δ_2 confirm that the membership and the VS algorithms indeed run in parallel. The VS algorithm, implemented in the project for this thesis, is more efficient (at least by $\approx 50\%$). In fact, δ_2 is comparable with the network round-trip time.

Although the absolute values of δ_1 are higher. This time is not in the scope of this thesis implementation, rather it is the time the membership service requires. The membership messages and VS-wsession messages propagate simultaneously as they should. The large values of δ_2 on Pentium II are due to the membership events generated on this machine. VS-wsession on Pentium II receives a start-change message from the local membership server right after a membership event is generated. However, in order to collect all synchronization messages, the Pentium II VS-wsession has to wait for: the membership information to propagate to other servers, the servers to send the start-change messages to their clients, and for the synchronization messages from other VS-wsessions to reach the Pentium II VS-wsession. Therefore, it takes longer.

The results confirm that the major overhead of the VS client is associated with the view transition. However, the synchronization algorithm is faster than the membership protocol.

The synchronization time is close to the network round-trip time, and it is not the limiting factor in the combined algorithm. If the network is stable, the message delivery overhead is negligible on a LAN – it is even less significant on a WAN. For a long stable communication periods, the cost of the view synchronization can be significantly amortized.

Chapter 7

Conclusion

The group communication system developed in this project is a useful tool in distributed fault-tolerant system development. The semantics that the VS client supports have proven to be useful. The performance is mainly constrained by the underlying network. Therefore, the scalability of the entire system in large depends on the message size and complexity. The current system has improved a fair amount of the GCS scalability characteristics due to the separation and parallelization of the membership and the VS algorithms. It has also implemented a more efficient VS algorithm, which minimizes the number of communication rounds. The complete **Xpand** system can now serve as a powerful application development infrastructure and as a base for further improvements and optimizations.

7.1 Future Developments

Several future enhancements are planned. The VS client can be extended to support the Optimistic Virtual Synchrony semantics (OVS) [12]. OVS allows the user application to send messages while in a reconfiguration state, where the current implementation blocks. This would improve network utilization and performance. Based on the information provided in the start-change message, the OVS client can send messages optimistically before the next view is determined by the membership service. If the start-change information is not completely accurate, the optimistic messages will be dealt with according to the policy

specified by an application.

Structured Virtual Synchrony[8] is a way to improve scalability of the current VS system. The Structured Virtual Synchrony, just like the membership service, exploits the hierarchical approach.

There are two main components in this hierarchical implementation: *controllers* and *clients*. Each client runs a simple algorithm that sends and receives messages from the client's controller. Each controller is responsible for a group of children clients; the controllers in turn communicate among themselves to share stability and client liveness information. In other words, the controller group implements a full-power virtually synchronous algorithm separately from the local clients, just as the membership servers run the membership algorithm in the membership group. Since the controllers are synchronized and communicate with their children clients via an RFIFO network layer, the local clients also provide the VS semantics to the application layer. Possibly, **Xpand**'s

The membership service relays both the controller and the client membership information to the controllers, which use it to implement virtual synchrony and in turn to inform the clients of their new views. The SVS architecture suggests the membership service be split in two groups. The first group will play its usual role in the controller group, whereas the second membership group will be distributed among the controllers in order to track the membership events within controllers' local groups.

The SVS approach increases scalability dramatically [8]. Since the controller group does not have to grow very fast with the number of clients, the network load also remains low compared to the standard VS algorithm. This means that the same physical networks can support a larger number of clients. Moreover, the number of the SVS levels can be increased. For example, some applications may need a higher degree of scalability and have more than two levels: controllers, sub-controllers and local clients. However, as the hierarchy increases it requires more complex controllers and more sophisticated membership service. A performance gain is still achievable with a moderate increase in complexity. Therefore, SVS is considered the next step in the development of the VS systems.

Bibliography

[1] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version: Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[2] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In Marios Mavronicolas, Michael Merritt, and Nir Shavit, editors, *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. American Mathematical Society, 1998.

[3] T. Anker, G. Chockler, I. Shnaiderman, and D. Dolev. The Design of Xpand: A Group Communication System for Wide Area Networks. Technical Report 2000-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, July 2000.

[4] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[5] K. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[7] R. Friedman and A. Vaysburg. High-performance replicated distributed objects in partitionable environments. Technical Report 97-1639, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, July 1997.

[8] Katherine Guo, Werner Vogels, and Robbert van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, September 1996.

[9] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 356–365, April 2000. Full version: MIT Technical Memorandum MIT-LCS-TM-593.

[10] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 344–355, April 2000. Full version: MIT Lab. for Computer Science Tech. Report MIT-LCS-TR-794.

[11] Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.

[12] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. In *19th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 2000. To appear. Previous version: Technical Report MIT-LCS-TR-792 MIT Lab for Computer Science; and Technical Report CS1999-634 University of California, San Diego, Department of Computer Science and Engineering.

[13] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report CS99-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, September 1999. Also Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, Laboratory for Computer

Science and Technical Report CS0964, Computer Science Department, the Technion, Haifa, Israel.