The Spread Wide Area Group Communication System

Yair Amir and Jonathan Stanton

Department of Computer Science
The Johns Hopkins University
{yairamir, jonathan}@cs.jhu.edu

Abstract

Building a wide area group communication system is a challenge. This paper presents the design and protocols of the Spread wide area group communication system. Spread integrates two low-level protocols: one for local area networks called Ring, and one for the wide area network connecting them, called Hop. Spread decouples the dissemination and local reliability mechanisms from the global ordering and stability protocols. This allows many optimizations useful for wide area network settings. Spread is operational and publicly available on the Web.

1. Introduction

There exist some fundamental difficulties with high-performance group communication over wide-area networks. These difficulties include:

- The characteristics (loss rates, amount of buffering) and performance (latency, bandwidth) vary widely in different parts of the network.
- The packet loss rates and latencies are significantly higher and more variable then on LANs.
- It is not as easy to implement efficient reliability and ordering on top of the available wide area multicast mechanisms as it is on top of local area hardware broadcast and multicast. Moreover, the available best effort wide area multicast mechanisms come with significant limitations.

Because of these difficulties, traditional work from the group communication community did not provide adequate solutions for wide area networks, even years after good solutions for local area networks were developed. The bulk of the work today that addresses wide area network settings comes from the networking community, starting from best effort IP Multicast and building up reliability services and some ordering with much weaker semantics than group communications.

This paper, which describes the Spread group communication system, is definitely coming from the group communication community's point of view. It implements, however, techniques and design features that are not that different from the techniques used to provide reliability over IP-Multicast, and scales dissemination and flow-control to wide-area networks. Since a group communication system has extensive, accurate knowledge of the system, the protocols used can be more precise and deliver better performance than many general networking protocols. The improvement, however, comes with a cost. Spread does more work per node and cannot scale with the number of users. However, it can be deployed to the wide area and can scale with the number of groups since no state needs to be maintained in the network routers for each group.

The Spread group communication system addresses the difficulties encountered in wide area networks through three main structural design issues:

- Spread allows for different low level protocols to be used to provide reliable dissemination of messages, depending on the configuration of the underlying network. Each protocol can have different tuning parameters applied to different portions of the network. In particular, Spread integrates two low-level protocols: one for local area networks called Ring and one for the wide area network connecting them, called Hop.

- Spread uses a daemon-client architecture. This architecture has many benefits, the most important for wide-area settings is the resultant ability to pay the minimum necessary price for different causes of group membership changes. Simple join and leave of processes translates into a single message. A daemon disconnection or connection does not pay the heavy cost involved in changing wide area routes. Only network partitions between different local area components of the network requires the heavy cost of full-fledged membership change. Luckily, there is a strong inverse relationship between the frequency of these events and their cost in a practical system. The process and daemon membership correspond to the more common model of "Lightweight Groups" and "Heavyweight Groups"[RG98].
- Spread decouples the dissemination and local reliability mechanisms from the global ordering and stability protocols. This decoupling allows messages to be forwarded on the network immediately despite losses or ordering requirements. This also permits pruning, where data messages are only sent to the minimal necessary set of network components, without compromising the strong semantic guarantees provided by typical group communication systems. In particular, Spread supports the Extended Virtual Synchrony model [MAMA94].

Spread is very configurable, allowing the user control over the type of communication mechanisms used and the layout of the virtual network. Spread provides priority channels to the application. Priority channels give expedited service to messages sent using them while preserving the ordering guarantees requested. We believe this is the first priority service implemented in a group communication system. Finally, Spread supports open-group semantics where a sender does not have to be a member of the group in order to multicast to it. These semantics have been found to be very useful in practice.

Although this paper is the first attempt at publishing the design and protocols, the Spread group communication toolkit has been available for awhile, and has been used in some research and in practical projects. The toolkit supports cross-platform applications and has been ported to several Unix platforms as well as Windows and Java environments.

Related Work

This work evolves out of the authors' previous work on the Totem and Transis group communication systems, as well as other research in multicast group communication systems and IP-Multicast related research.

Group communication systems in the LAN environment have a well developed history beginning with the ISIS [BR94] system, and more recent systems such as Transis [ADKM92], Horus [RBM96], Totem [AMMAC95], RMP [WMK94], and Newtop [EMS95]. These systems explored several different models of Group Communication such as Virtual Synchrony [BJ87] and Extended Virtual Synchrony [MAMA94]. Newer work in this area focuses on scaling group membership to wide area networks [ACDK98].

A few of these systems have added some type of support for either wide-area group communication or multi-LAN group communication. The Hybrid paper [RFV96] discusses the difficulties of extending LAN oriented protocols to the more dynamic and costly wide-area setting. The Hybrid system has each group communication application switch between a token based and symmetric vector based ordering algorithm depending on the communication latency between the applications. While their system provides a total order using whichever protocol is more efficient for each participant, Hybrid does not handle partitions in the network, or provide support for orderings other then total. The Multi-Ring Totem protocol [AMMB98] is an extension of the single-ring Totem protocol that allows several rings to be interconnected by gateway nodes that forward packets to other rings. This system provides a substantial performance boost compared to the single-ring Totem protocol on large LAN environments, but keeps the basic assumptions of low loss rates and latency and a fairly similar bandwidth between all nodes that limit its applicability to wide-area networks.

The totally ordered multicast protocol, SCALATOM [RGS98], scales in three ways: the protocol is only executed on those processes that are receiving the message, the message size scales with the size of the destination group, and it supports open groups. This protocol does not explicitly consider the issues of a heterogeneous wide-area environment, and requires a large number of messages to be sent to order each message and thus is probably unsuitable for a high latency, limited bandwidth setting like wide-area networks.

The Transis wide-area protocols Pivots and Xports by Nabil Huleihel [H96] provide ordering and delivery guarantees in a partitionable environment. Both protocols are based on a hierarchical model of the network, where each level of the hierarchy is partitioned into small sets of nearby processes, and each set has a static representative who is also a member of the next higher level of the hierarchy. Messages can be contained in any subtree if the sender specifies that subtree.

IP-Multicast is being actively developed to support Internet wide unreliable multicasting and to scale up to millions of users. Many reliable multicast protocols which run over IP-multicast have been developed, such as SRM [FJLMZ97], RMTP [LP96], Local Group Concept (LGC) [Hofm96], and HRMP [GG97].

The development of reliable multicast protocols over IP-Multicast has focused on solving scalability problems such as Ack or Nack implosion and bandwidth limits, and providing useful reliability services for multimedia and other isochronous applications. Several of these protocols such as SRM, have developed localized loss recovery protocols. SRM uses randomized timeouts with backoff to request missed data and send repairs, which minimizes duplicates, and has enhancements to localize the recovery by using the TTL field of IP-Multicast to request a lost packet from nearer nodes first, and then expand the request if no one close has it. Several other variations in localized recovery such as using administrative scoping and separate multicast groups for recovery, are also discussed in [FJLMZ97]. Other reliable multicast protocols like LGC use the distribution tree to localize retransmits to the local group leader who is the root of some subtree of the main tree. Spread has additional information about the exact dissemination of messages and where copies are buffered and so can use more precise local recovery that gets the packet from the nearest source.

HRMP [GG97] is a reliable multicast protocol which provides local recovery through the use of a ring, while wide area recovery is done through an ack tree. This is similar to our design that also uses a ring protocol for local reliability and dissemination while building a multicast tree for wide area reliability and dissemination. This work analyzes the predicted performance of such a protocol and shows it to be better then protocols utilizing only a ring or a tree. Our work validates and provides experimental support for their conclusions.

2. System Architecture

The Spread system is based on a daemon-client model where generally long-running daemons establish the basic message dissemination network and provide basic membership and ordering services, while user applications linked with a small client library can reside anywhere on the network and will connect to the closest daemon to gain access to the group communication services. The use of this model, as opposed to having the membership and ordering services provided by every client application, is very important in a wide area setting because the daemons minimize the number of membership changes (which are expensive) that the system must carry out across wide-area links, and provide a basic stable infrastructure on which to build effective wide-area routing, reliability, and ordering.

The cost of using a two level daemon-client model is that interference between different client applications using the same daemon configuration is possible, as discussed in [RG98]. This is minimized by the ability to run multiple daemon configurations each serving separate applications, and by our dissemination model(which only sends data messages to those daemons that need them and minimizes the cost of extra active daemons the application is not using). We do pay the cost, however, of additional context switches and inter-process communication. Overall, we think that the benefits outweigh the costs.

Spread is highly configurable, allowing the user to tailor it to their needs. Spread can be configured to use just one daemon in the world or to use one daemon in every machine running group communication applications. The best performance when there are no faults is achieved when a daemon is on every machine, while using fewer daemons decreases the cost of recovery. In principle, the Spread daemons can even be placed on the network routers where, for a cost in memory (buffers) on the router, the reliability of 'important' messages can be improved.

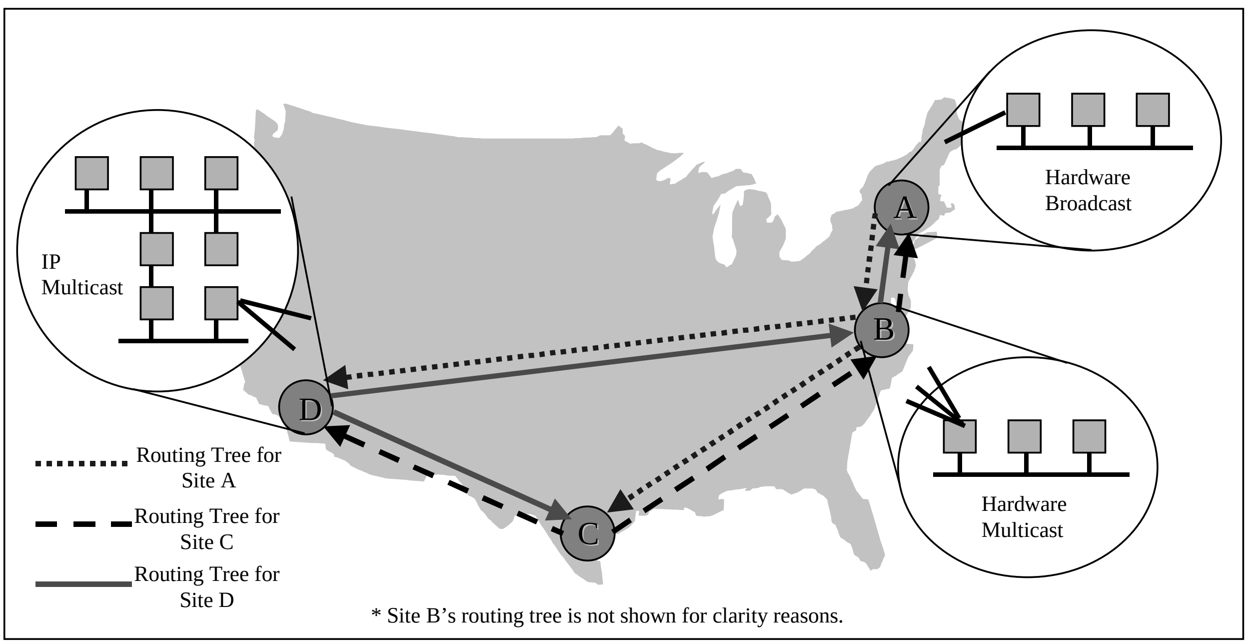

Figure 2.1: A Wide Area Network Configuration with Spread.

A sample network is given in figure 2.1, where several distinct sites are shown, geographically dispersed, with different cost links between them. In this paper a site is defined as a collection of machines which can potentially reach the other machines by one message, e.g. hardware broadcast, hardware multicast, or IP-Multicast. Each site can have up to several tens of machines on it, which does not impact the scalability of the Spread system since all operations not local to this site scale with the number of sites, not the total number of machines involved. All daemons participating in a Spread configuration know the complete *potential* membership when started, but all knowledge of the actual membership of active daemons is gathered dynamically during operation. Each site has one daemon that acts as the representative of the site, participating in the wide area dissemination. This representative is determined based on the current membership of the local site and is not hardwired.

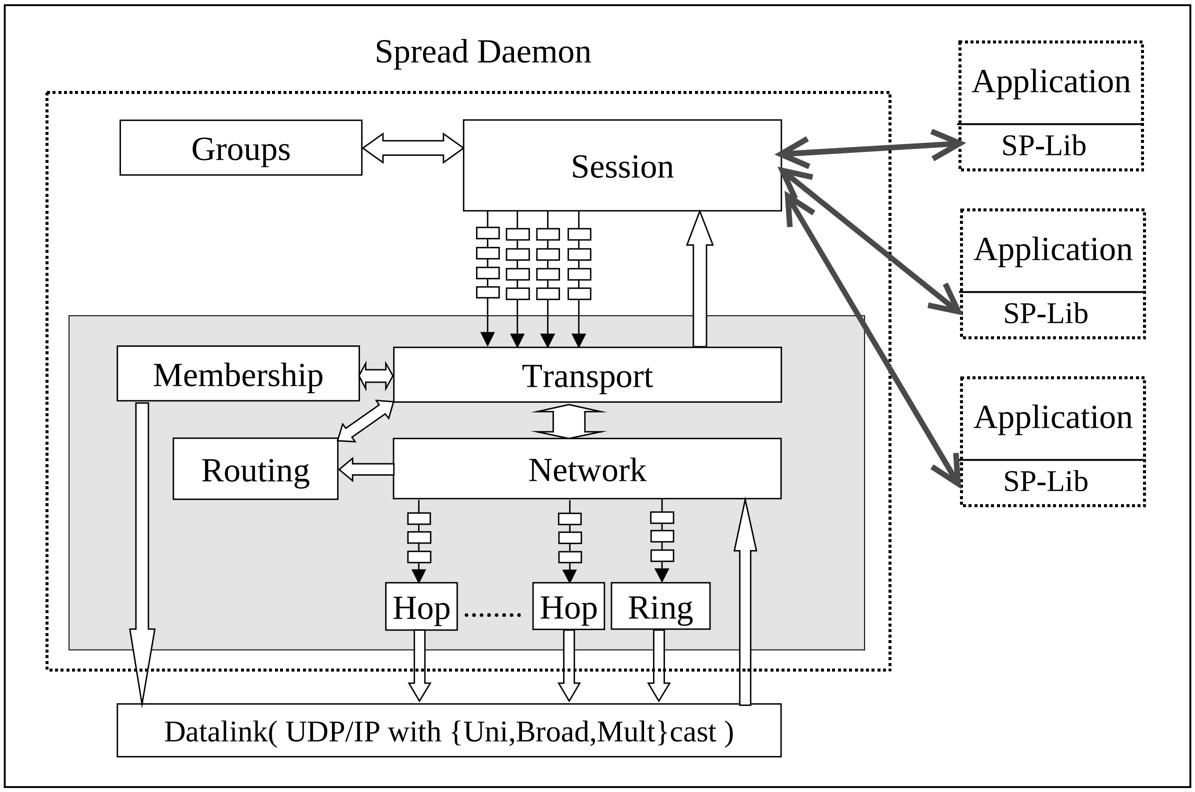

Figure 2.2: The Spread Architecture.

The Spread architecture is presented in Figure 2.2. User applications link with the SP_lib (or use the Java class) which provides the entire client interface described below. The connection between the SP_lib and the daemon is a reliable point-to-point connection, either IPC or through the network. The Session and Groups modules manage user connections, manage process group memberships, and translate daemon membership changes into process-group membership changes.

The shaded area in Figure 2.2 depicts the internal protocols of the daemon. The details of some of these protocols are described in Section 3. The highlights are:

- Multiple queues exist between the Session and Transport modules, one for each session. This allows priority handling.
- The Routing module computes the routing trees based on data from the Network module. The Transport module consults the Routing module to determine the links (Hops and Ring) on which each message should be sent.
- Several instantiations of the Hop module may exist, each of which represents one edge that is used by one or more routing trees.
- At most one Ring module provides dissemination and reliability on the local site if more then one daemon is active in that site.
- The Hop and Ring instances may be destroyed or instantiated as the membership changes.

Spread supports the Extended Virtual Synchrony model [MAMA94] of group membership. EVS can handle network partitions and re-merges, as well as joins and leaves. It provides several types of reliability (Unreliable, Reliable), ordering (Unordered, FIFO, Causal, Agreed), and stability (Safe) services for application messages. The implementation of site and daemon membership is not discussed in this paper.

All of the global orderings and stability (Causal, Agreed, Safe) are provided across all groups. If two messages are sent by different clients to different groups, anyone who has joined both groups will receive the two messages in the ordering guaranteed, even though they are received in different groups. FIFO ordering is provided with each connection to a daemon acting as a separate FIFO source for ordering purposes. As with the global orderings, the FIFO order is preserved across groups. In wide area networks, Reliable delivery becomes useful because, in principle, it will have no latency penalty compared to unreliable delivery because it can be delivered as soon as it is received. FIFO delivery becomes useful because a message will only be blocked from being delivered if messages prior to it **from the same application connection** are missing.

```
SP_connect( char *spread_name, char *private_name, int priority, int group_membership
             mailbox *mbox, char *private_group )

SP_disconnect( mailbox mbox )

SP_join( mailbox mbox, char *group )

SP_leave( mailbox mbox, char *group )

SP_multicast( mailbox mbox, service service_type,
               char *group,
               int16 mess_type, int mess_len, char *mess )

SP_receive( mailbox mbox, service *service_type, char sender[MAX_GROUP_NAME],
             int max_groups, int *num_groups, char groups[][MAX_GROUP_NAME],
             int16 *mess_type, int *endian_mismatch,
             int max_mess_len, char *mess )

SP_error( int error )
```

Figure 2.3: The Spread Application Programming Interface.

The complexity in spread is hidden behind a simple but complete programming API, which can be used both for LAN oriented services or WAN services without application changes, and which provides a clear model of group communications. The spread API is shown in Figure 2.3. An application can be written using only 5 functions (SP_connect, SP_join, SP_leave, SP_multicast, SP_receive) while the complete API allows more advanced features such as scatter-gather sends and receives, multi-group sends, polling on a connection, or comparing group ids. This API is based on the experience gained in the Transis project as well as later experience with Spread.

3. Protocols

3.1 Overview

The core of the spread system is the dissemination, reliability, ordering and stability protocols. Two separate protocol layers provide these services in Spread.

- Network layer - this layer is comprised of two components:
  - ⇒ Link-level protocols that provide reliability and flow control for packets. Spread implements two protocols, the Hop protocol for point-to-point connections, and the Ring protocol for multicast domains. Each protocol is optimized for its target domain.
  - ⇒ Routing that constructs the network out of Hops and Rings based on the current membership of connected daemons and its knowledge about the underlying network. The constructed network implements a different dissemination tree rooted at each site.

- Transport layer - this layer provides message delivery, ordering, stability, and global flow control. The Transport layer operates among all the active daemons in the system.

Building a routing tree rooted at each site is important for several reasons. It is clear that having an optimized tree per source site is more efficient than a shared tree. Since Spread is not meant to scale beyond several tens of sites (each containing up to several tens of daemons) the cost of computing these trees and storing forwarding tables is manageable. The benefit is enormous, especially since state and routing information is only maintained at end nodes.

It is important to remember that the overhead of building these trees can be amortized over the long lifetime of the site membership. This is in contrast with many multicast routing protocols which assume that the tree should only be built as needed since group membership changes are very common and so the cost of constructing a tree must be paid for very quickly, before the group membership changes in a way to make it invalid.

To utilize the underlying network as efficiently as possible, it is necessary to send full packets as much as possible. In order to utilize different packet sizes and to be able to pack multiple user messages or control packets into one network packet, all link level protocols actually act not on packets but on abstract "objects" which can vary in size from 12 bytes up to about 700 bytes. Each packet that is sent on the network is packed with as many objects as will fit. To improve readability, all the protocols described below are in terms of packets. In practice, however, the link level reliability and flow control are done per object.

3.2 Packet Dissemination and Reliability

The most basic service of any comprehensive multicast communication protocol is the dissemination of data messages to all receivers interested in them. An application level message sent to a group can range in size from 0 bytes of data up to 128 KB. Spread will both fragment and reassemble large messages to fit into the normal packet size of the underlying network, and pack small messages into full packets as much as possible without introducing unacceptable latency.

Spread builds multicast dissemination trees with each site in the active membership forming one node on the tree, either leaf or interior. A key observation made in general purpose multicast routing is that stability of the routing trees is very important to achieve workable, reliable routing. The spread systems separation of membership into three levels enables core routing to be based on a very stable configuration of sites, while individual application membership in groups can be highly dynamic. The actual creation of metrics to decide how to connect the sites into the most efficient trees is being investigated. Currently the trees are built by applying Dijkstra’s shortest-path algorithm to the current dynamic membership, with the underlying complete graph with weights being defined statically.

Once the routing trees have been constructed, forwarding of data packets within the trees is based on three principles:

- Non-Blocking: packets are forwarded despite the loss of packets ordered earlier.
- Fast-Retransmits: the immediate parent of the link where the loss occurred handles retransmission of lost packets.
- Pruning: packets are not forwarded to child links of the tree which have no members interested in the packet.

The non-blocking and fast retransmit properties are provided by the Hop protocol described below, while the pruning is provided by the routing lookup code which filters out any child links where all the sites below those links have no member in the groups the packet is destined for. The resolution of which sites are interested in a packet is made when the packet is created at the source site. The decision is slightly conservative in that possibly some non-interested sites may get a packet, but every site that is interested is guaranteed to get it. This comes into effect during the period when an application has asked to join or leave a group but the join or leave has not gone into effect yet. To provide the ordering and stability guarantees, aggregated control information is sent to all sites.

Packet dissemination, reliability and flow control is provided by the two link-level protocols - the Hop protocol for point-to-point connections and the Ring protocol for multicast and broadcast domains, as described below.

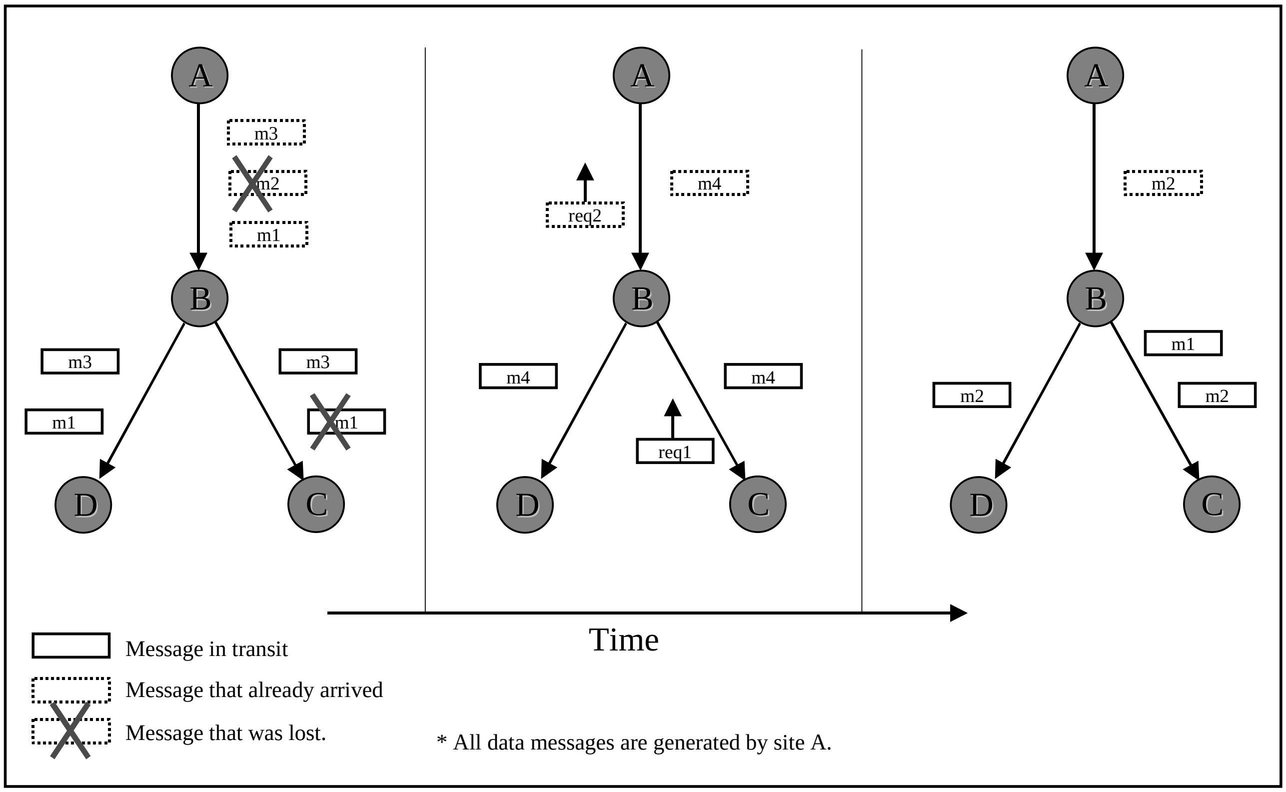

Figure 3.1: A Scenario Demonstrating the Hop Protocol.

The Hop Protocol

The Hop protocol operates over an unreliable datagram service such as UDP/IP. The core idea of the hop protocol is to provide the lowest latency possible when transferring packets across networks consisting of several hops by handling the loss of packets on a hop-by-hop basis instead of end-to-end and by forwarding packets immediately even if they are not in order. In cases where it is possible to run the Spread protocol in the routers that connect sites involved in group communications, the advantages of this would be even higher.

The hop protocol uses negative acknowledgements to request retransmission of packets and positive acknowledgements to determine up to what sequence value can be removed from the sender's buffers. Flow control is provided by a token/leaky bucket regulator that limits the number of packets that can be sent at one time and limits the maximum overall rate. In addition, a sliding window, which limits the total outstanding packets on the link, is used to prevent arbitrarily long delays in getting missing packets. Figure 3.1 demonstrates the Hop protocol.

A hop link is bi-directional, so each side can send to the other. The sender side uses the following local variables:

s_highest_linkseq	The highest link_seq attached to a packet so far.
s_other_end_aru	The link sequence aru reported by the other end of the link.
S_cur_window	The current size of the sliding window used for flow control.

The receiver has these variables:

r_highest_seq	The highest sequence value seen on this link so far.
r_link_aru	The sequence value up to which all packets have been received.
Retransmit list	A linked list of sequence values which have been missed and are waiting for retransmit by the sender.

The last two variables are used by both the sender and receiver sides:

Pkt_cnt_linkack	The number of packets sent or received since the last link ack was sent.
max_pkt_btw_ack	A tuning value limiting how many packets can be sent or received before sending a link ack.

Three types of packets are sent across a hop link:

- Data : This is a portion of a user data message or an internal spread message created by a higher layer.
- Ack: This is a copy of the receivers current link_aru value and the senders highest_linkseq.
- Nack: This is a list of all sequence values which the receiver has missed and should be resent.

Both acks and nacks are limited to the link on which they originate.

During normal operation the sender side of a hop link will send data packets and an occasional link ack, and will receive link acks from the receiver side. This continues as long as the receiver side keeps up with the sender to within the size of the sliding window and no packets are lost.

When the receiver gets a packet with a higher sequence then (r_highest_seq + 1), then it has missed some packets and so it adds all the sequence values between r_highest_seq and the received object sequence to the retransmit list. It then sets a timer to wait for a short time to see if the missing packets were just delayed or reordered in the network, if the timer expires and there are still packets in the retransmit list then a link nack is sent requesting those packets. The link nack will be resent with the current list of missing packets every time the timer expires until all missing packets have been received. The receiver does keep a count of how many times each sequence value has been requested to be resent. If that count gets above a certain value then the receiver declares the sender to dead and initiates a membership change. This membership change occurs even if the receiver has received other data from the sender as it is very important to eliminate this "failure to receive" problem, otherwise, the system will be forced to block eventually due to buffering.

When the sender receives a link nack it adds the requested packets to the queue of data to be sent to the receiver and deletes the nack. The retransmitted packets will be included as sent data in the flow control calculations. Figure 3.2 presents the pseudo code for the Hop protocol.

The Ring protocol

The Ring protocol is used when there is more then one active daemon in a site. Note that a site is a collection of machines which can potentially reach the other machines by one message, e.g. hardware broadcast, hardware multicast, or IP-Multicast. Each member has a unique identifier that is static over crashes and restarts. The Ring protocol is a modification of the ring protocol used in

```
Hop_Send_Data(linkid)
    Update token bucket for hop
    while( token is available in bucket for hop )
         Each packet is assigned a unique link_seq value and stored in Link[linkid].open_pkts.
         Send(linkid, pkt)
    if( still_data_available for this hop )
           Event_Queue(Hop_Send_Data, linkid, SendTimeout)

Hop_Send_Ack(linkid)
    ack.link_aru := Link[linkid].r_link_aru
    ack.link_max_sent := Link[linkid].s_highest_linkseq
    Send(linkid, ack)
    Event_Queue(Hop_Send_Ack, linkid, LongHopAckTimeout)

Hop_Send_Nack(linkid)
    add all link_seq values found in retransmit list to nack.req_list[]
    Send(linkid, nack)
    Event_Queue(Hop_Send_Nack, linkid, HopNackTimeout)

Hop_Recv()
    recv(pkt)
    case(pkt.type)
    ACK:
        if (Link[linkid].s_other_end_aru < ack.link_aru)
           remove packets with linkseq < ack.link_aru from Link[linkid].open_pkts[]
           Link[linkid].s_other_end_aru := ack.link_aru
        if (Link[linkid].r_highest_seq < ack.link_max_sent)
           add sequence numbers from r_highest_seq to link_max_sent to retransmit list
           Event_Queue(Hop_Send_Nack, linkid, HopNackTimeout)
    NACK:
        for (i from 0 to nack.num_req-1 )
            req_linkseq := nack.req_list[i]
            pkt := Link[linkid].open_pkts[req_linkseq % MAXPKTS]
            queue_packets_to_send(linkid, pkt)
    DATA_PKT:
        Link[linkid].pkt_count++;
        if (Link[linkid].pkt_count == 1)
            Event_Queue(Hop_Send_Ack, linkid, ShortHopAckTimeout)
        if (Link[linkid].pkt_count > Max_Count_Between_Acks)
            Send_Link_Ack(linkid)
        if (msg_frag.link_seq == Link[linkid].r_highest_seq + 1)
            /* Right on time packet without drops */
            if (Link[linkid].r_link_aru == Link[linkid].r_highest_seq)
                Link[linkid].r_link_aru++
                Link[linkid].r_highest_seq++
        else if (msg_frag.link_seq <= Link[linkd].r_highest_seq)
             /* Duplicate or delayed packet */
             remove msg_frag.link_seq from retransmit list and update lowest_missing_linkseq
             if duplicate packet
                 return Null
             if (lowest_missing_linkseq == NONE)
                 Link[linkid].r_link_aru := Link[linkid].r_highest_seq
                 Event_Dequeue(Hop_Send_Nack, linkid)
             else if (msg_frag.link_seq < lowest_missing_linkseq)
                 Link[linkid].r_link_aru := lowest_missing_linkseq - 1
        else
             /* we missed some packets before this one */
             add seq numbers from r_highest_seq to msg_frag.link_seq to retransmit list
             Link[linkid].r_highest_seq := msg_frag.link_seq
             Event_Queue(Hop_Send_Nack, linkid, HopNackTimeout)
        return packet
    Otherwise:
        return packet

* Upper layers will call Event_queue( Hop_Send_Data, Linkid, 0) when data has been queued to send.
```

Figure 3.2: The Hop Protocol.

Totem[AMMAC95] and Transis[A95]. In Totem and Transis, the Ring protocol is used to provide reliability, global flow control and global ordering.

Spread uses the ring protocol for one main purpose: packet-level reliability and flow control within the local site, and one secondary purpose: message-level stability within members of the ring. The crucial point is that the same token is used for both these functions. In the same token circulation, the ring aru calculation algorithm updates both the packet and message aru fields. So, we are almost not paying anything in complexity and latency of the protocol to get message-level knowledge. In contrast to Totem and Transis, global ordering and flow control are provided by our transport layer protocol described below. This limits the usage of the ring protocol to tasks for which it is most effective: local area ordering, reliability, and dissemination.

In the extreme, the entire collection of daemons can be configured as one site connected by routed IP-multicast. This configuration will not take advantage of our wide-area reliability protocols and can have poor performance.

The rotating token has the following fields:

Type	Regular except during membership changes.
Link_seq	The highest sequence number of any reliable packet sent on the ring.
Link_aru	The sequence number of which all reliable packets have been received up to by all members of the ring. It is used to control when a link can discard any local references to a packet.
Flow_control	A count of the number of packets sent to the ring during the last rotation of the token, including retransmits.
rtr list	A list of all the sequence values that the previous token holder is asking to be retransmitted.
Site_seq	The highest sequence number of any reliable **message** originating on this ring. This sequence is local to the site and combined with the site_id provides a unique identifier of every **message** sent in the system.
Site_lts	The highest LTS value seen by any member of the ring so far. This is used to provide a causally consistent ordering for Agreed and Safe messages.
Site_aru	The LTS number up to which all members of the site have received all messages with an LTS value lower then this.
Site_aru_modifier	The identifier of the site member that last modified the site_aru value.

Upon receiving the token a daemon will handle any retransmits requested by the previous token holder, then process messages received from client applications, send packets up to the limit imposed by flow control, then update the token with new information and send it to the next daemon in the ring. After sending the token the daemon will attempt to deliver any messages it can to the client applications. For each message processed into the system, a new site_seq and site_lts value is assigned and the counters are incremented. For every reliable packet sent, a unique link_seq is assigned and the counter is incremented.

To update the link and site aru values on the token, a daemon compares the local aru values with those on the token, if the local value is less, then the token value is lowered to be the local value and the site_aru_modifier field is set to the id of the daemon. If the local value is equal to or higher then the token value then the daemon raises the token value only if the site_aru_modifier is this daemons id, or the site_aru_modifier is zero, indicating that no daemon has lowered it during the last round. All members of the ring can thus calculate the highest aru value they can locally use by taking the lesser of this just calculated token aru value, and the token aru value from the previous round of the token.

The ring provides flow control by limiting the number of packets each member can send during each rotation of the token. The number of packets which can be sent on the entire ring per round, and a limit on how much each individual member can send per round are tuning parameters. The daemon simply sends the lesser of its individual limit and the total limit minus the value in the flow_control field plus what it sent last time. Figure 3.3 presents the pseudo code for the Ring protocol.

```
Ring_handle_token(token)
     drop token if it is malformed, a duplicate, or the wrong size.
     if( Link[linkid]->highest_seq < token.link_seq ) Link[linkid].highest_seq := token.link_seq;
     answer any retransmit requests which are on the token
     update SiteLTS and SiteSeq values
     Assign site SiteLTS and SiteSeq values to new application messages
     Calculate flow control for this ring
     Send_Data(linkid)
     update tokens flow control fields
     add any link_seq values I am missing to the token req_list[]
     update Link[linkid].my_aru
     update token.link_aru, token.set_aru, token.link_seq, token.site_seq, token.site_lts,
            token.site_aru, token.site_aru_modifier
     send token to next daemon in the ring
     calculate Link[linkid].ring_aru based on token.link_aru and last_token.link_aru
     discard all packets with link_seq < ring_aru from Link[linkid].open_pkts[] array
     calculate site_aru (Highest_ARU[my site]) based on token.site_aru and last_token.site_aru
     copy token to last_token
     Deliver_Mess()
```

Figure 3.3: The Ring Protocol.

3.3 Message delivery, ordering and stability

The Transport layer, which is the upper layer of the protocol, provides the required guarantees for message delivery, ordering and stability service semantics. The Transport layer uses the Network layer for dissemination, local flow control, and packet reliability. A message in Spread will typically pass over several links while being sent to the other daemons. In the presence of faults the transport protocol at each daemon keeps a copy of all complete messages sent or received until they become stable across the entire system, and thus can provide retransmissions of messages which were missed during the membership change triggered by the faults. The recovery protocols in the case of faults are not discussed in this paper.

Ordering and stability are provided by a symmetric transport level protocol run by all active daemons. It uses unique sequence values assigned to every message originating at a site, the unique id assigned to every site, and a logical clock defined by Lamports "happened before" relation[L78], to provide a total order on all messages in the system, and to calculate when messages become stable. Stability is defined as a message which this daemon knows that all relevant daemons have received, and thus can be removed from the system.

Every message is assigned the following values before dissemination:

- A site id.
- A site sequence.
- A Lamport Time Stamp (LTS).
- A session sequence.

Hence, a message can be completely ordered based strictly on information contained in the message. The use of logical clocks and sequence values has the substantial benefit of being completely decentralized and of making recovery from partitions and merges in a manner consistent with EVS possible. Without born-ordered messages, Agreed and Safe delivery cannot be achieved across network partitions.

Unreliable messages have a separate session sequence because they need none of the support provided to reliable messages. Since they could be dropped, we do not want their loss to interfere with the reliability protocols. The reason they have a sequence value at all is to make sure the daemon does not deliver duplicate copies of a message to the application and to have a way of identifying a specific unreliable message so the daemon can reassemble it from packets. If some portion of an unreliable message does not arrive within a specific time of the first piece of the message which arrived, then all portions which did arrive are discarded. Unreliable messages are still sent only when a daemon has the token on the local ring because of flow control issues. Unreliable messages are delivered as soon as the full message arrives and do not take part in the main delivery protocol described below.

Reliable messages use the network layer reliability. Each link guarantees reliable transport within a bounded time, absent processor or network faults. Thus end-to-end reliability is provided in the case where there are no faults because eventually every packet will make it across all the hops and rings to all the daemons which need it. Reliable messages are delivered as soon as the complete message is received since they do not provide any ordering guarantees. Therefore, their delivery is never delayed due to other messages.

FIFO messages provide the same reliability guarantee as reliable messages. They also guarantee they will be delivered after all messages from the same session with lower session sequence values. To provide per-session FIFO ordering, Spread incurs the cost of a small amount of memory per session. In an active system with many sessions this cost is small compared to the required message buffers and may yield considerable benefit to message latency.

Agreed messages are delivered in order consistent with both FIFO and traditional Causal ordering. This order is consistent across groups as well. To provide the Agreed order, the local daemon delivers messages according to a lexicographic order of the {LTS, site id} fields of the message. The cost is that to totally order a message the daemon must have received a message or pseudo update message from every other site with values that indicate no message with a lesser {LTS, site id} will ever be transmitted. Spread minimizes this cost because it only requires a message from every site, not every daemon. The only potentially faster method requires a sequencer that introduces a centralized resource. Further, a sequencer cannot provide EVS in the presence of network partitions [MAMA94].

Safe messages are delivered in order consistent with Agreed ordering. Stability is determined in a hierarchical manner. Each site uses the Site_aru field of the local ring to generate a site-wide All-Received-Upto value. This value represents the local site stability status. This ARU value is then propagated to all other sites using the network layer. The minimum of the ARU values from all of the sites determines the global message stability. Safe messages equal to or below this value can be delivered and all delivered messages equal to or below this can be discarded.

Spread employs several techniques to optimize the calculations required by the above orderings, such as hash tables, multi-dimensional linked lists and caching. The details of these optimizations are beyond the scope of this paper due to lack of space.

4. Implementation and practical experience

The Spread system is implemented as a daemon written in ANSI C and POSIX 1003.x specified interfaces and has been ported to a number of Unix systems (Solaris, Irix, BSDI, Linux, SunOS) and Windows (95 and NT), and a library written in both C and Java which is linked with the client application (all in all, about 20,000 lines of code). The library supports multi-threaded client applications and is quite small at 25-55 Kbytes depending on the architecture. Several sample applications and documentation for the system are also provided.

We currently support two versions of the Spread daemon. The first is a robust production version that is publicly available on the Web. The second is a research version that is described in this paper. Both versions share the same library interface and the same service semantics. Applications may use them interchangeably.

The protocols in the publicly available version of Spread are more limited in their support of the wide area settings then described in this paper. In particular, they employ an adapted ring structure with direct point-to-point dissemination between the sites (rather than multiple trees). The research version is not as robust as the production version although the protocols described in this paper are all implemented in it. We use this version as a test bed for current experiments with membership and flow control.

We have conducted some experiments over the Internet to test the correctness of the implementation and get some preliminary measurements of performance. Our testbed consisted of three sites, Johns Hopkins University in Maryland, University of California at Santa Barbara, and Rutgers University at New Jersey, connected over the public Internet. The sites consisted of machines ranging from Sparc-5 to Ultra-2 workstations running the Solaris operating system, and Pentium Pro and II workstations running Linux and BSDI operating sytems. During the tests the machines were also under normal user load, and no changes were made to the operating system.

On average, conducting several experiments each one multicasting 10,000 1 Kbyte messages to receivers located at every site, the throughput of the system was 755,300 bits per second. The values varied only slightly depending on which site was the source and which were the receivers. This is logical because in our network all of the optimal dissemination trees happen to be the same. When the routing was configured non-optimally with UCSB forwarding between Hopkins and Rutgers the throughput showed a 20-50% decrease.

To evaluate the gains achieved by the Spread architecture, we also ran the same tests with the production version of Spread which is described above. With this version the average throughput on the same network was 159,387 bits per second. This amounts to a gain of 4.7 times for the new architecture. Another way to evaluate the benefit of the new architecture was to look at a configuration of two sites, Hopkins and UCSB, connected across the Internet. This isolates the limitations of a ring architecture across high latency links. In this case the difference was even greater: 1,149,511 bits per second vs. 215,907 bits per second, which amounts to a 5.3 times improvement. These experimental results reinforce the observations in [GG97].

Other experiments showed that the local area network and the wide area links can support different transfer rates since their dissemination and reliability is decoupled. Thus, local only traffic can be supported at higher rates then wide area traffic. Also, experiments showed different hop links supported widely varying throughput. As a result, pruning can be very effective. On local area networks both versions of Spread achieve excellent performance, comparable to the basic Transis and Totem protocols. We note that these experiments are only preliminary before we have had adequate chance to tune the new protocols.

The Spread toolkit has been used in several research projects, a prototype military system, and several classes of undergraduate and graduate students. Spread was used to record connectivity data between several machines connected at different locations on the Internet. This data, gathered over six months, was used to evaluate different quorum systems in a practical setting [AW96]. Spread serves as the group communication subsystem in the Walrus Web replication system [Shaw98]. Under Walrus, a single Web server is replicated to several clusters of identical server, where each cluster resides in a different part of the Internet. Spread is utilized to disseminate load information between the Web servers as well as to track the membership of them. This allows Walrus to balance the load between the different clusters. Researchers at the USC Information Studies Institute are using Spread to implement group security protocols [STW98].

The JHU Applied Physics Lab, in collaboration with the Naval Surface Warfare Center (NSWC), is using Spread in an advanced prototype of a combat control system named HiPer-D. The main reasons for using group communication in HiPer-D are two fold: efficient reliable multicast dissemination, and synchronized state transfer. Both enable fault tolerance services that can maintain real-time response to events even when the system experiences server failures. This project uses the cross-platform support in Spread to inter-operate with Solaris and NT machines.

5. Conclusion

We have presented the Spread wide area group communication system. The Spread architecture allows for different link level protocols. A Ring protocol for LANs and a Hop protocol for WAN are integrated together with site based routing trees and pruning to provide a high performance reliable dissemination network. Efficient global ordering and stability protocols were implemented on top of this network. Preliminary experiments validate the usefulness of the techniques utilized in the system. Spread is operational and publicly available on the Web, and has been used in several research and development projects inside and outside our group.

Bibliography

[A95] Amir, Y. 1995. *Replication using Group Communication over a Partitioned Network*. Ph.D Thesis., Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel.

[ACDK98] Anker, T., Chockler, G. V., Dolev, D., and Keidar, I. 1998. Scalable Group Membership Services for Novel Applications. To appear in *the DIMACS book series, proceedings of the workshop on Networks in Distributed Computing*. Edited by: Marios Mavronicolas, Michael Merritt, and Nir Shavit.

[ADKM92] Amir, Y., Dolev, D., Kramer, S., and Malki, D. 1992. Transis: A communication sub-system for high-availability. *In Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, IEEE Computer Society Press, Los Alamitos, CA, 76-84.

[AMMB98] Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M., and Budhia, R. K. 1998. The Totem Multiple-Ring Ordering and Topology Maintenance Protocol. *ACM Transactions on Computer Systems* 16, 2 (May), 93-132.

[AMMAC95] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. 1995. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*. 13, 4 (Nov.), 311-342.

[AW96] Amir, Y., Wool, A. 1996. Evaluating Quorum Systems over the Internet. *In Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing*, June, 26-35.

[BJ87] Birman, K. P., and Joseph, T. 1987. Exploiting Virtual Synchrony in Distributed Systems. In *11th Annual Symposium on Operating Systems Principles*, Nov, 123-138.

[BR94] Birman, K. P., and Van Renesse, R. 1994. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, March.

[EMS95] Ezhilchelvan, P. D., Macedo, R. A. and Shrivastava, S. K. 1995. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems* (Vancouver, Canada, May/June). IEEE Computer Society Press, Los Alamitos, CA, 296-306.

[FJLMZ97] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L. 1997. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, *IEEE/ACM Transactions on Networking*, 5, 6 (Dec.), pp. 784-803.

[GG97] Gu, L., Garcia-Luna-Aceves, J.J. 1997. New Error Recovery Structures for Reliable Networking. In *Proceedings of the Sixth International Conference on Computer Communications and Networking* (Los Vegas, Nevada, Sept.)

[H96] Huleihel, N. 1996. *Efficient Ordering of Messages in Wide Area Networks*. Masters Thesis. Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel.

[Hofm96] Hofmann, M. 1996. A Generic Concept for Large-Scale Multicast. In *Proceedings of International Zurich Seminar on Digital Communications* (Zurich, Switzerland, Februrary). Springer Verlag.

[L78] Lamport, L. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21, 7, 558-565.

[LP96] Lin, J.C., Paul, S. 1996. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of IEEE INFOCOM*. March. 1414-1424.

[MAMA94] Moser, L. E., Amir, Y., Melliar-Smith, P. M., and Agarwal, D. A. Extended Virtual Synchrony. In *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems* (Poznan, Poland, June). IEEE Computer Society Press, Los Alamitos, CA, 56-65.

[RBM96] Van Renesse, R., Birman, K. P., and Maffeis, S. 1996. Horus: A flexible group communication system. *Communications of the ACM* 39, 4 (Apr.), 76-83.

[RFV96] Rodrigues, L. E. T., Fonseca, H., Verissimo, P. 1996. A dynamic hybrid protocol for total order in large-scale systems. Selected portions published in *Proceedings of the 16th International Conference on Distributed Computing Systems* (Hong Kong, May).

[RGS98] Rodrigues, L.E.T., Guerraoui, R., Schiper, A. 1998. Scalable Atomic Multicast. To be published in *Proceedings of the Seventh International Conference on Computer Communications and Networking*. (Lafayette, Louisiana, Oct.)

[Shaw98] Shaw, D. 1998. *Walrus: A Low Latency, High Throughput Web Service Using Internet-Wide Replication*. Masters Thesis, Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland.

[STW98] Steiner, M., Tsudik, G., and Waidner, M. 1998. Cliques: A New Approach to Group Key Agreement. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, (Amsterdam, The Netherlands, May), 380-387.

[WMK94] Whetten, B., Montgomery, T., and Kaplan, S. 1994. A High Performance Totally Ordered Multicast Protocol. In *Theory and Practice in Distributed Systems, International Workshop*, LNCS 938, (Sep.)