

IMPLEMENTATION AND PERFORMANCE TESTING OF A GOSSIP-BASED COMMUNICATION SYSTEM

Kim P. Kihlstrom, Joel L. Stewart, N. Tobias Lounsbury, Adrian J. Rogers, and Michael C. Magnuson

Department of Computer Science, Westmont College

955 La Paz Road, Santa Barbara, CA, USA 93108

{kimkihls, jstewart, nlounsbury, adrogers, mimagnus}@westmont.edu

ABSTRACT

Computer systems today are increasingly complex, employing vast numbers of nodes connected over large geographic areas. As these types of systems become more pervasive and integral to a variety of services, there is a heightened need for scalability and reliability in the underlying communication system.

We present an implementation of a gossip-based, wide-area group communication system that is resilient to process and communication failures and is scalable to a very large number of nodes. We provide implementation details and describe our performance tests. The results are presented and analyzed.

KEY WORDS

Scalability, reliability, gossip-based probabilistic multicast

1 Introduction

Computer systems have become highly distributed, employing large numbers of nodes connected over long distances, and extremely complex. As such systems become more pervasive and integral to a variety of services, there is a heightened need for scalability and reliability in the underlying communication system. Reliable communication that is scalable is important to support a variety of distributed applications and Web Services, including distributed databases and publish-subscribe systems.

A group communication system provides delivery guarantees for messages sent to a group of nodes in a system. Such delivery guarantees can include reliable delivery to all operational nodes, as well as source ordering, causal ordering, and/or total ordering of messages despite node and link failures.

The size, distribution, and complexity make these systems difficult to design, configure, analyze, characterize, and maintain. Great care must be taken to ensure consistency even in the presence of various faults and failures that occur. Additionally, large systems include inherent asynchrony due to variable and unpredictable processing and transmission delays. Many current protocols for group communication suffer dramatic performance loss when the number of nodes increases.

Starblab is an implementation of a peer-to-peer gossip-based communication protocol as described by Ker-

marrec, *et al.* [1]. In a gossip protocol, each node maintains a partial view of the group membership, to which it forwards (“gossips”) the messages it receives. Gossip-based communication protocols have a number of desirable properties, such as scalability to very large numbers of nodes and resilience to node and communication faults. The communication protocol relies on a self-organizing group membership protocol developed by Ganesh, *et al.* [2]. The membership protocol operates in a completely decentralized manner, providing members with a partial view size that is appropriate for the size of the system but without the need for any node to know the group size.

Each node in the system maintains both an *inView* of nodes from which it receives gossip messages and an *outView* of nodes to which it forwards messages. The *outView* of each node provides a partial view of the entire system. Whenever a message is received at a node, that message is then forwarded to all the nodes in the *outView*. The communication protocol provides probabilistic guarantees that messages will be delivered atomically, *i.e.*, that the messages will reach every node in the system.

We have implemented the Starblab group communication protocol and conducted performance tests on our local network and on Emulab [3], an integrated experimental environment for distributed systems and networks.

Our performance results for Starblab indicate that latency is directly proportional to the number of hops that a message travels. Pittel [4] showed that, for a system organized in the manner we describe, the number of hops required to reach all the nodes (*i.e.*, the diameter of the system) is proportional to $\log(n)$ in a system of size n . Because the measured latency in our system is proportional to the number of hops traveled, the latency for a multicast to reach all nodes in the system is proportional to $\log n$ and thus scales gracefully.

The rest of this paper is organized as follows. We outline related work in Section 2 and describe our implementation in Section 3. We discuss our performance testing and results in Section 4, and we give conclusions and plans for future work in Section 5.

2 Related Work

Gossip protocols scale very well to large numbers of nodes. Our implementation is based on a probabilistic gossip-

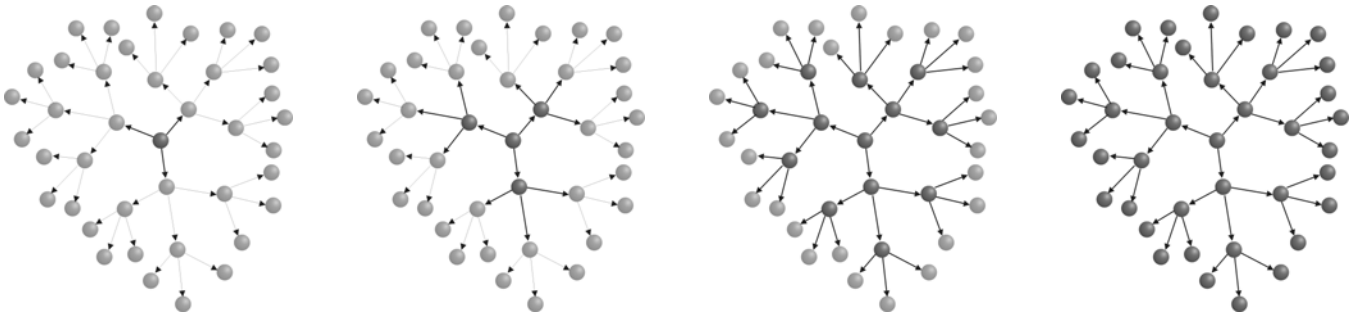


Figure 1. Dissemination of a message in a gossip protocol.

based group communication protocol described by Kermarrec, *et al.* [1]. In the gossip protocol, each node maintains a partial view of the group membership, to which it forwards (“gossips”) the messages it receives. The communication protocol has a number of desirable properties, such as scalability to a very large number of nodes and resilience to node and communication faults.

Our work is also based on SCAMP (Scalable Membership Protocol) [2], a self-organizing group membership protocol developed by Ganesh, *et al.* The membership protocol operates in a completely decentralized manner, providing members with a partial view size that is appropriate for the size of the system but without the need for any node to know the group size.

The focus of the preceding work [1, 2] is on theoretical analysis and simulation results, rather than on an implementation. Thus, our contribution is in providing an implementation and performance results.

Other work has also explored gossip-based communication protocols. Birman, *et al.* [5] described Bimodal Multicast, a probabilistic protocol that uses peer-to-peer interaction for reliable multicast. Bimodal Multicast is designed for a static set of processes that communicate over a fully connected point-to-point network, in contrast to the dynamic, partially connected system we describe.

Eugster and Guerraoui [6] developed a scalable gossip-based multicast algorithm that ensures, with high probability, that a process interested in a multicast event delivers that event and a process not interested in that event does not receive it.

While gossip-style protocols scale very well, other group communication protocols are designed for a relatively small system and do not scale well to a large number of nodes.

SCALATOM [7] is a scalable fault-tolerant algorithm that ensures total order delivery of messages sent to multiple groups of processes. The ordering algorithm uses two companion protocols, a reliable multicast protocol and a consensus protocol, which are not required to use the same communication channels or to share common variables with the total order protocol.

The Spread system [8] is based on a daemon-client architecture where long-running daemons establish the mes-

sage dissemination network. This allows effective wide-area routing. Spread services include reliable message delivery and total ordering even in the event of node failures and network partitions.

The MAFTIA middleware architecture [9] supports multi-party interactions reliably under a group oriented middleware suite. MAFTIA uses a network system consisting of LANs situated on a WAN to support wide-area stability.

3 Implementation

We have implemented a gossip-based group communication protocol. A gossip protocol relies on a peer-to-peer communication model. The protocol is quite scalable to very large systems, and is particularly suitable for systems that are not completely connected, or in which some nodes and links may fail and recover. The load is distributed among the nodes, providing scalability. Resilience to node and link failures is achieved by using a redundancy of paths and messages.

The basic communication protocol operates in much the same way as gossip spreads or an infection is disseminated, as shown in Figure 1. When a node p needs to communicate a message m , it sends the message to a randomly chosen subset of nodes. When a node q receives m for the first time, it delivers the message and forwards it to a randomly chosen subset of nodes. The communication protocol provides probabilistic guarantees that a message will be delivered atomically, *i.e.*, that the message will reach every operational node in the system. Pittel [4] showed that, for a gossip-based system such as the one we describe, the number of gossip rounds required to reach all the nodes in a system of size n is proportional to $\log(n)$.

The group communication system we describe includes a gossip protocol for distributing a message, a membership protocol that determines the subset of nodes to which a node gossips a message, and fault handling mechanisms to deal with events such as node and link failures. The organization of the group communication protocol is shown in Figure 2.

In the rest of this section we describe the system model, the gossip protocol, the membership protocol, the

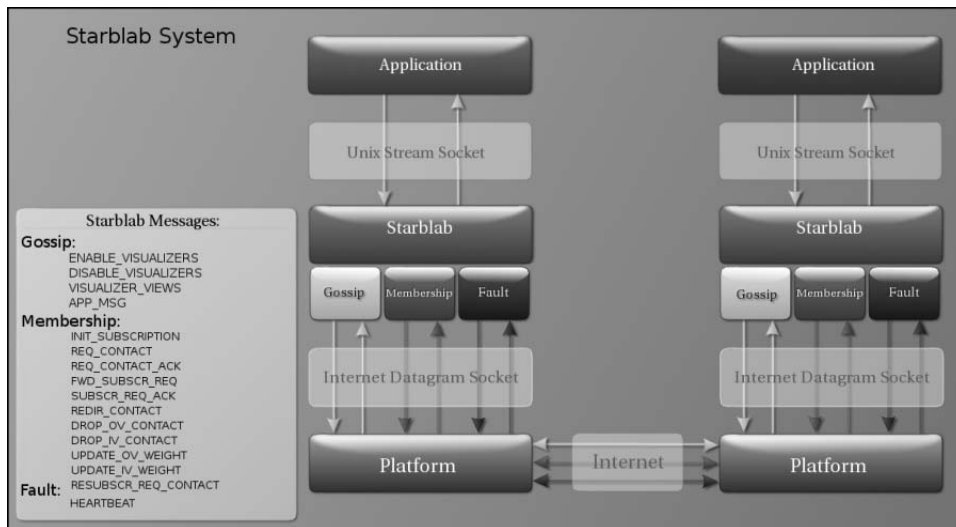


Figure 2. Organization of the communication system.

fault handling mechanisms, and the details of the implementation.

3.1 System Model

We consider a system consisting of n nodes, where n may be very large, *e.g.*, tens of thousands. No node needs to be aware of the total number of nodes in the system. The underlying graph formed by the nodes and the links between nodes is assumed to be connected in that every node is reachable by at least one path from every other node, where a path consists of one or more links. However, a given node is not assumed to have direct communication with all of the other nodes, *i.e.*, the underlying graph is not completely connected. Some of the nodes may be faulty and cease to send messages. Communication between processors is unreliable, and thus messages may need to be retransmitted and may be delayed arbitrarily.

We distinguish among the terms *generate*, *send*, *receive*, *forward*, and *deliver*. A *regular* message is *generated* by the application to be *sent* on the communication medium. A processor that *receives* a message from the communication medium can *forward* it to other processors, and/or *deliver* it to the application.

3.2 Gossip Protocol

The protocol makes use of several types of messages as listed below:

- APP_MSG:** A regular message that has been generated by the application for delivery back to the application
- INIT_SUBSCRIPTION:** An initial subscription request that is sent by a subscribing node to a proxy node
- REQ_CONTACT:** A message containing the identifier of a subscribing node. Initially sent by the proxy node as

part of the indirection mechanism. Will be forwarded until the count is zero, at which point the receiving node functions as the contact node

- REQ_CONTACT_ACK:** A message that is sent from a new contact node to the newly subscribing node. It notifies the subscribing node that the sender is its contact. The subscribing node will add the contact node to its *outView* and *inView*. The subscribing node sets a lease timer upon receipt of this message
- FWD_SUBSCR_REQ:** A message containing the identifier of a subscribing node. Initially sent by the contact node and forwarded until the subscription is accepted
- SUBSCR_REQ_ACK:** A message that is sent to a subscribing node from a node that has accepted its subscription. The receiving node adds the new node to its *inView*
- REDIR_CONTACT:** A message that is sent from an unsubscribing node to a node in its *inView*. The receiving node will drop the unsubscribing node from its *outView* and replace it with the identifier contained in the message payload
- DROP_OV_CONTACT:** A message that is sent from a re-subscribing node to a node in its *inView*. The receiver node will remove the sender node's contact data from its *outView*
- DROP_IV_CONTACT:** A message that is sent from an unsubscribing node to a node in its *outView*. The receiving node will remove the sender node's contact data from its *inView*
- UPDATE_OV_WEIGHT:** A message that contains a weight update for the arc between the receiving node and the sending node
- UPDATE_IV_WEIGHT:** A message that contains a weight update for the arc between the receiving node and the sending node

RESUBSCR_REQ_CONTACT: A message that is sent from a resubscribing node to a node in its *outView*. The receiving node will forward the subscription request to all the members of its *outView* and add the resubscribing node to its *outView*

HEARTBEAT: A message that is sent periodically by a node to aid in fault detection

The fields¹ of a message *m* are:

type: message type; one of the types listed above

sender: identifier of the sender of *m*

seq: sequence number for an APP_MSG, or \perp for all other types

payload: application-generated contents of an APP_MSG, number of times to forward for a REQ_CONTACT message, weight for UPDATE_OV_WEIGHT or UPDATE_IV_WEIGHT message, node identifier for a REDIR_CONTACT message, or \perp for all other types

As described by Kermarrec, *et al.* [1], each node *p* in the system maintains both an *inView_p* of nodes from which it receives gossip messages and an *outView_p* of nodes to which it forwards messages. The *outView_p* of each node provides a partial view of the entire system. Whenever a message is received at a node for the first time, that message is then forwarded to all the nodes in the *outView_p*.

3.3 Membership Protocol

The gossip protocol depends on a membership protocol to form and maintain the *inView_p* and *outView_p* at each node *p*. As described by Ganesh, *et al.* [2], the membership protocol includes mechanisms for a node to join the system and for the underlying graph to be rebalanced. The membership protocol is executed autonomously at each node, without global knowledge of the system.

A new node *r* requests to join the system by sending an INIT_SUBSCRIPTION message to a well-known node *p*, called a proxy node. The proxy node *p* begins a mechanism called indirection. Indirection is designed to ensure that the initial contact node is chosen approximately at random from all the nodes in the system. To begin the indirection mechanism, the proxy node selects a counter value and sends a REQ_CONTACT message containing this counter to a node *q* in its *outView_p*. The counter determines how many times the REQ_CONTACT message is to be forwarded. The node *q* to which the REQ_CONTACT message is sent is chosen from the *outView_p* with probability based on a weighting mechanism described in the original protocol [2].

When node *q* receives a REQ_CONTACT message, it checks the counter value. If the counter is greater than zero, node *q* decrements the counter and forwards the REQ_CONTACT message to a node that is chosen from its

outView_q with probability based on the weighting mechanism [2]. If the counter in the REQ_CONTACT message is zero, node *q* serves as the contact node for the subscribing node *r* and sends a REQ_CONTACT_ACK to *r*.

The new node *r* begins with only its contact *p* in its *outView_r*. Node *p* then sends a FWD_SUBSCR_REQ message to all nodes in its *outView_p*. The FWD_SUBSCR_REQ message contains the identifier of the subscribing node *r*. Additionally, *p* creates extra copies of the FWD_SUBSCR_REQ message and forwards each to a random node in its *outView_p*. The number of extra copies is referred to as *extra_p* and is a design parameter that determines the level of fault-tolerance of the system.

When a node *p* receives a FWD_SUBSCR_REQ message containing the identifier of subscribing node *r*, and *r* is not already in *p*'s *outView_p*, *p* will, with a given probability, keep the FWD_SUBSCR_REQ message and add *r* to *p*'s *outView_p*, or else forward the FWD_SUBSCR_REQ message to a random node in *p*'s *outView_p*. The probability of keeping the FWD_SUBSCR_REQ message is inversely related to the size of *p*'s *outView_p*, and serves to keep the system balanced in the size of its *outViews* as well as its diameter, both of which are $O(\log(n))$.

The protocol makes use of an additional lease mechanism to ensure that the graph is balanced, *i.e.*, that the *outViews* are of the required size. This lease mechanism is described in Section 3.4, and also serves to reduce the likelihood that a node is isolated for a prolonged period of time. The membership protocol also employs a mechanism for a node to unsubscribe as described in the original protocol [2].

3.4 Fault Handling

While the membership protocol described creates a connected graph, it is possible that link and node failures can cause a node to become isolated. To rectify this situation, a node *p* periodically sends a heartbeat message to the nodes in its *outView_p*. A node that fails to receive any heartbeat messages for a sufficient amount of time will initiate a process of subscribing again called resubscription.

In addition, each subscription has a finite lifetime, called its lease. When a subscribing node receives a REQ_CONTACT_ACK message from its contact node, it sets a lease timeout. When the lease timeout expires, the node begins the resubscription process.

To resubscribe, a node *p* sends a RESUBSCR_REQ_CONTACT message to an arbitrary node *q* in its *outView_p*. The receiving node *q* will forward the RESUBSCR_REQ_CONTACT message to all the members of its *outView_q* and add the resubscribing node to its *outView_q*. The resubscribing node *p* also sends a DROP_OV_CONTACT message to every node *r* in its *inView_p*. The receiving node *r* will remove the sender node *p*'s contact data from its *outView_r*. The resubscribing node *p* will remove all nodes from its *inView_p*, but will not change its *outView_p*.

¹Throughout this paper, we use \perp to refer to an empty value. When an empty message is created, all of the fields contain \perp .

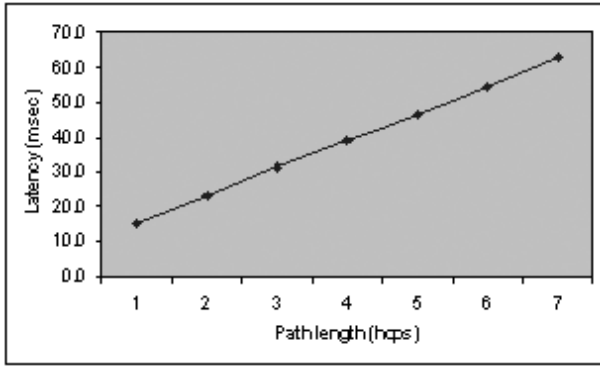


Figure 3. Latency of the group communication system as measured in local lab.

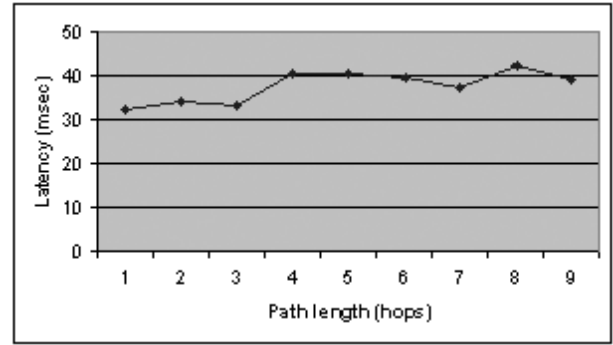


Figure 5. Latency of the group communication system as measured on Emulab.

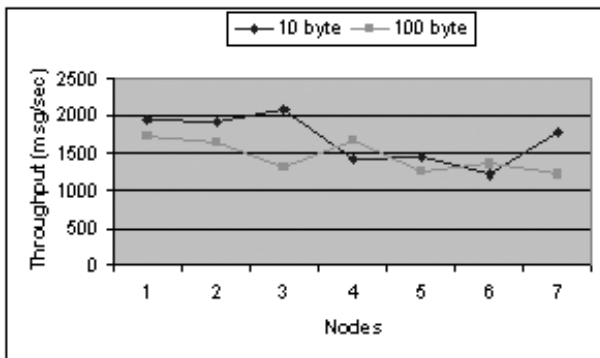


Figure 4. Throughput of the group communication system for 10 byte and 100 byte messages.

3.5 Implementation Details

The implementation consists of approximately 3,000 lines of C++ code. Our implementation is available for download from our website [10]. It is organized into modules, which include the main entry module as well as node, contact, and message classes.

The node class implements the main event loop and is multi-threaded. A node contains two sockets: an internet datagram socket for communication with other nodes, and a unix stream socket for internal communication with the applications residing on the local machine. A listener thread listens for messages on the sockets.

Execution begins in the main module and an instance of the node class is created. The node constructor initializes the sockets and threads. Control then passes to the main event loop. The node processes messages that are received and sends messages according to the protocol specifications.

4 Evaluation

The implementation has been tested in a lab consisting of eight workstations running SUSE 10.2, each with a single

2.4 GHz AMD Athlon 64 4000+ processor and 1 GB RAM, connected through a 100 Mbit/s Ethernet. Performance for the group communication protocol as measured in this lab is shown in Figures 3 and 4.

To measure throughput, a single node sends an initial message to all other nodes, and then begins generating and transmitting regular messages. Each receiver records the time t_1 when the initial message is received, checks the validity of each regular message received, counts valid messages until the desired number M is reached, and then records the time t_2 . The throughput is then calculated as $(t_2 - t_1)/M$.

The mean throughput in messages/second as a function of the number of nodes, for message payload sizes of 10 bytes and 100 bytes, is shown in Figure 4. These measurements were conducted with a single node sending to other nodes in the system.

The measurement of latency in a distributed system is complicated by the clock skew between the sending and receiving node. To perform latency measurements, we made use of a technique described by Budhia, *et al.* [11]. The sending processor p_1 appends its current clock value to a message being transmitted. The receiving processor p_2 reads its own clock value immediately after receiving the message. The difference between the two clock values is the message delivery latency L_i plus the clock skew S . Summing these latencies over M messages, processor p_1 measures

$$L_1 = \sum_{i=1}^M (L_i + S)$$

and processor p_2 measures

$$L_2 = \sum_{i=1}^M (L_i - S).$$

If M is large enough and the clock skew does not vary significantly over time, we can estimate the mean latency L as

$$L = \frac{L_1 + L_2}{2 * M}.$$

We sent $M = 100$ messages of size 18 bytes at a rate of one message per second for each measurement of average latency; this was repeated 30 times for each test run.

The mean latency in milliseconds as a function of the number of hops the message travels is shown in Figure 3; these results indicate that latency is directly proportional to the number of hops traveled. As indicated in Section 3, for a gossip-based protocol such as the one we describe, the number of hops required for a message to reach all the nodes in a system of size n (i.e., the diameter) is proportional to $\log(n)$. Because the measured latency in our system is proportional to the number of hops, the latency for a multicast is proportional to $\log n$ and is thus scalable to very large systems.

We also performed latency measurements over Emulab [3], an integrated experimental environment for distributed systems and networks. We performed the measurements using up to ten nodes running Fedora Core 6, connected by a 100 Mbit/s Ethernet. We sent $M = 100$ messages of size 18 bytes at a rate of one message per second for each measurement of average latency; this was repeated 30 times for each test run. The mean latency as a function of the number of hops the message travels is shown in Figure 5. As in the case of tests in our local lab, the latency measured on Emulab is shown to be directly proportional to the number of hops traveled, and thus the latency for a multicast is proportional to $\log n$, where n is the number of nodes in the system.

5 Conclusions and Future Work

With the growth of the internet and increased demand for web services has come a heightened need for group communication systems that are scalable to a large number of nodes distributed over a wide area.

We have implemented Starblab, a gossip-based group communication protocol. Gossip-based communication protocols have a number of desirable properties, such as scalability to very large numbers of nodes and resilience to node and communication faults.

Our main contribution is the implementation and performance testing of our system. We have shown that the latency for our implementation is proportional to $\log n$, where n is the total number of nodes in the system, and is thus scalable to very large systems.

Future work will include further performance testing, fault injection and anomaly detection measurements, as well as extending the implementation to render it intrusion-tolerant.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0534167.

References

- [1] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), 2003, 248–258.
- [2] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), 2003, 139–149.
- [3] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, USA 2002, 255–270.
- [4] B. Pittel. On spreading a rumour. *SIAM Journal on Applied Mathematics*, 47(1), 1987, 213–223.
- [5] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2), 1999, 41–88.
- [6] P. Th. Eugster and R. Guerraoui. Probabilistic multicast. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, 313–324.
- [7] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proceedings of the 7th IEEE International Conference on Computer Communications and Networks (IC3N'98)*, Lafayette, Louisiana, USA, 1998, 840–847.
- [8] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference On Dependable Systems and Networks*, New York, NY, USA, 2000, 327–336.
- [9] P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the IEEE 3rd Information Survivability Workshop*, Cambridge, MA, USA, 2000, 157–161.
- [10] Westmont College. *The Starfish Project Website*. <https://starfish.westmont.edu/>.
- [11] R. K. Budhia, L. E. Moser, and P. M. Melliar-Smith. Performance engineering of the Totem group communication system. *Distributed Systems Engineering*, 5(2), 1998, 78–87.