HyperCast:
A Protocol for Maintaining a Logical Hypercube-Based Network Topology

A Thesis

Presented to

The Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science (Systems Engineering)

by

Tyler Kien Beam

May 1999

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the
requirements for the degree of
Master of Science (Systems Engineering)

Tyler Beam

This thesis has been read and approved by the Examining Committee :

Thesis Advisor

Committee Chairman

Accepted for the School of Engineering and Applied Science :

Dean, School of Engineering and
Applied Science

May 1999

Table of Contents

Index of Figures

Index of Tables

Abstract

With the steadily growing size and popularity of the worldwide Internet, new applications such as tele-collaboration tools and videoconferencing are becoming more popular. These new networked multimedia applications require data transfer to a potentially very large number of recipients. IP Multicast is one technology that can help transmit data more efficiently to a large group of users. However, multicast transmission supports only unreliable service, meaning that it is not guaranteed that a message sent using multicast will actually get to its destinations. Therefore, for applications where reliability is required, scalable methods must be used to ensure reliability by retransmitting data when necessary. One general approach to this problem is to establish a hierarchy among group members, so that data repair operations can be localized. This organization of group members is a control topology.

This thesis presents a new protocol that we call *HyperCast* for the organization of group members, by using a logical hypercube as the hierarchy of computers. A hypercube structure as the control topology has many scalability advantages over existing methods. With these advantages, the hypercube structure provides an increase in the maximum potential number of users in reliable multicast groups over current methods, with little bandwidth wasted on protocol overhead.

A full implementation of the protocol written in Java is presented. The robustness of the protocol was exhaustively tested using a verification tool. Large-scale experiments using a computer cluster at the University of Virginia were conducted to quantitatively assess the performance of the protocol, as well as to demonstrate its scalability and the applicability of the protocol to large real-world uses. Additional uses of the hypercube structure are also discussed.

1 Introduction

Computer applications such as videoconferencing, shared document editors and collaborative tools place enormous demands upon the networking technologies connecting the computers [MAC94]. Video, audio, and collaborative applications often need to distribute information to many destinations at once. Efficiently sending data to a large number of recipients requires a different paradigm of thinking about information dissemination [DEE89]. This manner of data transmission needs to provide scalable communication for a large number of simultaneous users.

1.1 Comparison of Unicast and Multicast Transmission Methods

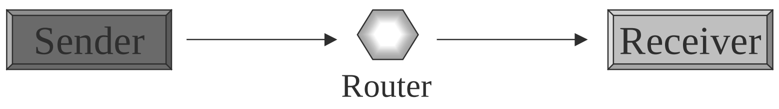


Figure 1. Unicast communication with a single receiver: Packets are sent only to the destination, via a router.

The most prevalent mode of data transfer in packet switching networks is *unicast* communication. One computer (the sender) transmits a bundle of information called a *packet* to another computer (the receiver) connected to it via the Internet [COM91]. In Figure 1, the transmission is seen going through an intermediate location, a *router*. Routers are specialized computers on the Internet that serve simply to forward packets to a destination, possibly via other routers. Note that only one copy of the information is sent between the sender and the receiver.

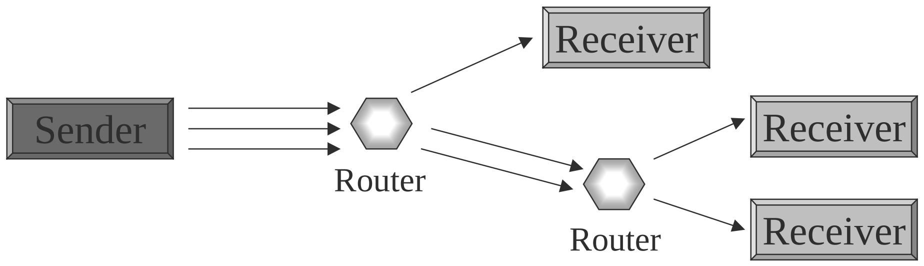


Figure 2. Unicast communication with multiple receivers: Redundant unicast packets are sent to intermediate routers.

Unicast communication serves the needs of many network applications, such as distributing electronic mail and serving web pages to web browsers. However, the limitations of unicast communication become apparent when it is applied to situations in which there is more than one receiver of the same data. As seen in Figure 2, the network bandwidth consumption at the sender is proportional to the number of receivers. The load placed on the sender to individually service each one of them becomes prohibitive as the number of receivers grows large [DEE91].

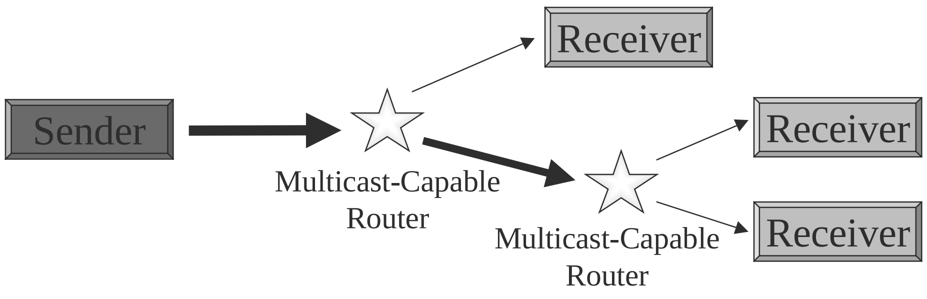


Figure 3: Multicast communication: Single multicast packets are sent to multicast-capable routers.

A more scalable method of data transfer to a set of receivers is to use *multicast* transmissions [DEE91]. Multicast transmissions differ from unicast transmissions in that they are addressed to a set of receivers rather than a single destination. The set of senders and receivers that exchange multicast packets for an application build a *multicast group*.

Multicast packets can then be sent to the group as a whole. When a multicast packet reaches a multicast-capable router, the destination multicast group of the packet is read. Based upon the multicast group membership, the multicast-capable router creates copies of the packet and sends the copies to their appropriate destinations (Figure 3).

Multicast-capable networks may be separated on the Internet by routers that are not capable of interpreting multicast packets. To solve this problem, multicast packets are encapsulated within unicast packets and sent through unicast *tunnels* between multicast networks. Multicast-capable hosts and routers together form a virtual network overlaid on top of the Internet called the Internet Multicast Backbone, or MBone [CAS94]. The connections between all of the multicast-capable routers and application endpoints define the *multicast distribution tree* for the group.

Since multicast transmission allows the sender to send each packet only once to the entire multicast group rather than having to send a packet to each receiver individually, multicast communication is a more scalable approach to group communications [DEE91].

However, using multicast transmissions is not a complete solution for all applications. The primary problem on which this research is focused is that multicast transmissions are *unreliable*, i.e. it is not guaranteed that data sent will actually be received by all (or even any) of the intended receivers. The unreliability of multicast transmissions is due to the fact that the underlying transport mechanism for multicast packets is the User Datagram Protocol (UDP), an unreliable service. In comparison, unicast applications can either use UDP or use the Transmission Control Protocol (TCP), a reliable service.

Packet loss frequency using IP Multicast has been measured as high as 25% [ZAB96]. For some applications, lack of reliable data transfer does not pose a problem. For example, when transmitting digital video a small percentage of lost video data does not have an adverse effect on the perceived quality at the receiver. A lost video frame may not be noticed by the user if subsequent frames are successfully transmitted.

For applications which require reliable data transfer, as in the case of file transfers between computers or document-based collaboration tools, a system between the members of the multicast group must be devised to handle detection and correction of lost or corrupted packets. The system of checks and procedures for retransmission of lost data is a reliable multicast protocol.

1.2 Problem Statement

Many different protocols exist which ensure reliability in multicast communications [LEV96A] [PIN94]. All such protocols have to address the following basic problem: If each receiver of a multicast group exchanges control information directly with the sender to ensure reliability, then the sender suffers from a load proportional to the number of receivers in the group. For example, if all receivers in a multicast group send an acknowledgement that a packet has been received (ACK) or a negative acknowledgement of a missing packet (NACK) directly to the sender, then the sender is deluged with messages [LEV98]. While ACKs and NACKs sent directly between the sender and receiver are standard practice in ensuring reliability in unicast transmissions (as in TCP), this method does not scale well to large numbers of receivers. This problem is called ACK/NACK implosion [LEV96A].

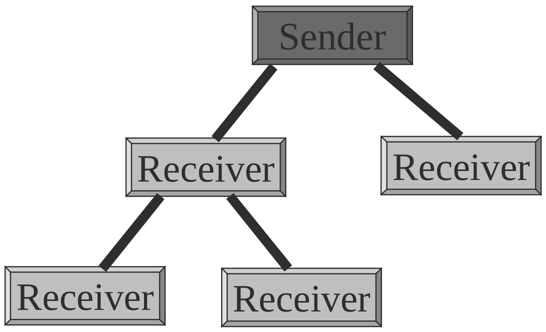


Figure 4. A logical tree topology can be used to aggregate NACKs.

One approach to the ACK/NACK implosion problem is to construct a hierarchy of multicast group members. One such hierarchy is a tree [RAM87], where each member of the multicast group is represented as a node in the tree, and the sender is at the root (Figure 4). A structure such as this is used so that a node reports packet losses only to its parent, instead of directly to the sender. The parent node then performs the retransmission of data if the data is available, or else aggregates the NACKs from its children and sends a single NACK up to its own parent. With this approach, the load of receiving retransmission requests and performing data repair operations is distributed across the whole tree, rather than relying on just the sender. This is a more scalable approach since the load at every node is proportional to its number of children, and not proportional to size of the receiver set.

Note that the tree structure is a *logical* organization that does not necessarily have to bear any correlation to the *physical* organization of the nodes on the network. In the discussion of the multicast distribution network in Section 1.1 the nodes and routers were tied by physical connections, however in this case the links between nodes are defined by logical relationships.

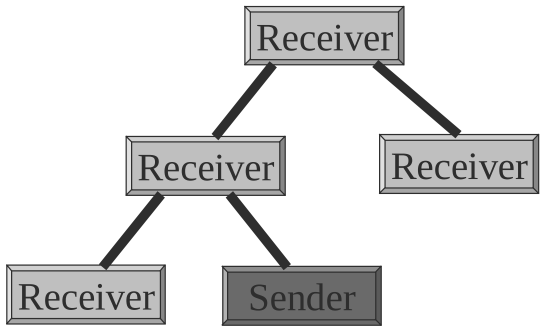


Figure 5. A tree topology has difficulties with multiple senders.

A tree control topology has good performance features for one-to-many distribution of data: a balanced tree distributes the load of retransmission evenly over the receiver set and has a short total path length [LEV96B]. However, the tree is also necessarily rooted around a single sender. Thus, a tree topology is useful for applications that have one source of data, however it is inefficient and difficult to use a tree structure if multiple nodes are sending data (Figure 5).

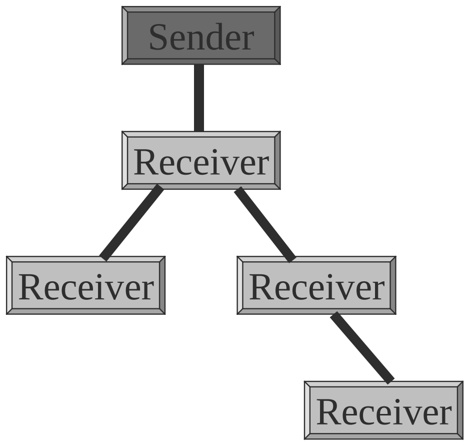


Figure 6: The example tree "re-hung" from the sender.

If a non-root node sends data, the tree must either be wholly reconstructed with the new sender at the root, or the existing tree links must be "re-hung" from the new sender (Figure 6). A single tree topology that is re-hung and used for multiple senders is known as a *shared tree* [LEV96B]. In Chapter 3, *K-ary* shared trees will be discussed, which are shared trees where each node can have up to *K* children.

If the tree is reconstructed, there is overhead associated with establishing new logical connections to create the tree rooted at the new sender. If the tree is re-hung, the resulting tree can be unbalanced and suffer from poor performance measures such as a long average path length (number of steps from each receiver back to the sender) and poor load-balancing across nodes [LIE98B]. Note in Figure 6 that the re-hung tree has a greater average path length than the original tree in Figure 5.

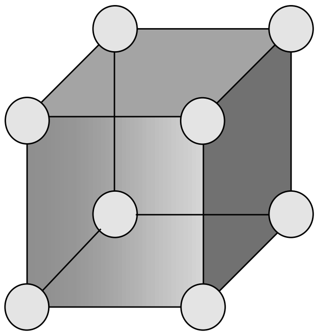


Figure 7. A hypercube control topology of dimension 3.

J. Liebeherr and B. S. Sethi proposed an alternative control topology based on a logical hypercube [LIE98A]. A hypercube is a generalization of a three-dimensional cube into *N* dimensions, also known as a measure polytope (Figure 7). It is an extension of a cube into *N*-space much like a cube is a three-dimensional extension of a square. An *N*-dimensional hypercube will have 2^N vertices and $N \cdot 2^{N-1}$ edges, and will be more formally defined in Chapter 3. Using this topology, group members are arranged as vertices of the hypercube, and the logical links between them lie along the hypercube edges.

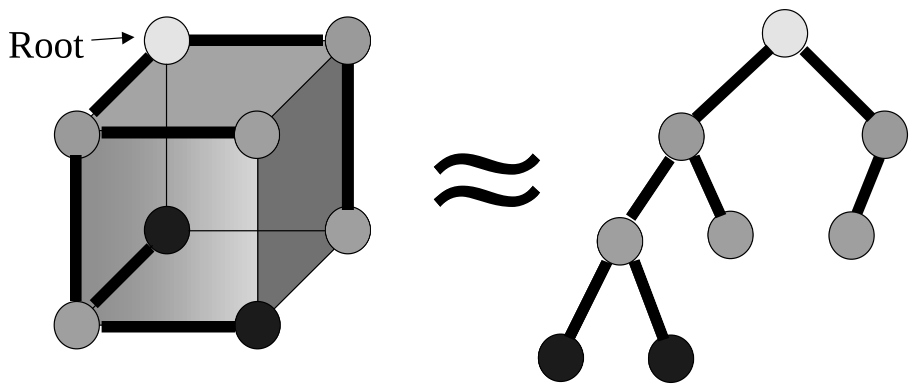


Figure 8. A hypercube can be used as a tree.

Using a hypercube has benefits over a control structure such as a tree, since it can be shown that a tree can be easily superimposed over the hypercube structure (Figure 8) [LIE98A]. Due to the relative symmetry of the logical hypercube, this superimposed tree can be rooted at any of its nodes. This means that using the hypercube has all the performance advantages of a tree topology, without the limitation of having best-case performance only when there is one sender.

The problem this research addresses is the design, specification, implementation and evaluation of the *HyperCast* protocol that creates and maintains a hypercube control topology from a set of group members. In order to perform efficiently with very large groups of nodes, the protocol must be able to organize nodes into a logical hypercube without any node having knowledge of the entire group. Because of real-world problems such as network faults and packet loss, the protocol must detect nodes that have failed unexpectedly and perform maintenance and repair of the hypercube structure.

2 Previous and Related Work

2.1 Classes of Reliable Multicast Protocols

In all computer networks where transmissions are subject to packet loss, the reliability of data transfer is a concern. At the most basic level, there are two primary ways by which reliability can be improved: forward error correction (FEC) and automatic-repeat-request (ARQ) [NON96].

FEC methods add redundancy into the data transmitted in order for the receiver to be able to recover the original information even with some packet loss [NON96]. For example, a trivial implementation of FEC is to transmit every packet twice over the network. With such redundancy, individual packet losses will most likely not have any adverse effect. In many cases, the advantages of having the receiver be able to completely recover from packet loss by itself can be shown to outweigh the bandwidth penalty incurred due to the redundant transmissions. FEC is a means of improving reliability, however it does not provide any guarantee of successful data transmission by itself. If packet loss rates approach 100%, the receivers will not have enough data to reconstruct the original information regardless of how much redundancy is encoded. Therefore FEC is used to improve reliability, but it cannot guarantee it.

ARQ schemes rely on retransmitting data when packet loss is detected. Retransmissions continue until the data has been transmitted successfully, thereby guaranteeing reliability. There are two main classes of ARQ schemes: *sender-initiated* and *receiver-initiated*.

In sender-initiated protocols, the sender bears responsibility for ensuring reliability for all receivers, by keeping explicit information about the set of receivers and verifying the delivery of data to each one. When the sender has received some form of acknowledgement of successful data reception from each of the receivers, then it can proceed with the knowledge that reliable transmission has been achieved.

A receiver-initiated scheme places the responsibility of lost packet detection upon the receiver. The receiver needs to request the retransmission of data when it detects packet losses. The sender cannot explicitly verify successful delivery of data. Instead, a lack of retransmission requests from the receivers is interpreted as an implicit sign of correct transmission. Packet losses can be detected by the receiver via the use of *sequence numbers*, which are consecutive numbers that the sender attaches to every packet that it transmits. If the receiver detects a gap in the sequence numbers that it has received, then packet loss has occurred.

2.2 Sender-Initiated Approaches

To ensure reliability in a sender-initiated approach the sender must have complete knowledge of the receiver set. This is due to the fact that the sender must keep track of state information for each of the receivers in the group in order to determine if packets have been reliably delivered to all receivers. Additionally, the receivers must have a mechanism to acknowledge packet reception to the sender. If each receiver sends an ACK back to the sender, the sender is subject to ACK implosion. Sender-initiated protocols must be designed to solve this scalability problem.

The Negative Acknowledgement with Periodic Polling (NAPP) protocol was developed as a broadcast protocol for LANs [RAM87]. With NAPP, the sender attaches sequence numbers to data packets that it transmits. Receiver nodes multicast NACKs back to the entire group’s multicast channel when packet losses are detected, and the sender replies by rebroadcasting the lost data. Note that a receiver cannot detect if it has lost the last set of packets that the sender transmitted, since the loss of those last packets does not create a gap in the receiver's list of received sequence numbers. Therefore the sender periodically polls the receiver set, requesting that each of the receivers transmit the sequence number of their last successfully received packet. The polling serves two purposes: (1) if the receiver's last successfully received sequence number conflicts with the sender's last transmitted sequence number, it can be used as a NACK for the lost packets, and (2) it also acts as an implicit acknowledgement of successful delivery of all the packets up to that point. The NAPP protocol is also notable because it was the first protocol to implement NACK *suppression*, a system used in many protocols that is explained here: If a receiver planned to broadcast a NACK for a lost packet but also had already received another node's NACK for that same packet, then it suppresses its own NACK response to reduce duplicate NACKs on the multicast channel. This avoids an implosion problem, since ideally only one NACK is broadcast per lost packet for the whole group.

The Xpress Transfer Protocol (XTP) is another sender-initiated protocol that makes use of this suppression technique [STR92]. XTP allows the application three different levels of reliability to choose from: fully reliable service, UDP-like unreliable service, and a mode in which receivers transmit a negative acknowledgement immediately when packet loss is detected. XTP also suggested the use of *slotting* and *damping* to reduce the scalability damage from ACK implosion. Acknowledgements are "slotted" by introducing a random delay before the transmission of an ACK, thereby reducing the likelihood of many receivers transmitting identical ACKs at the same time. Also, receivers "damp" their control messages by not transmitting them if they determine that their control messages are redundant with messages that other nodes have already broadcast. Slotting and damping is an extension of the suppression technique introduced by NAPP.

The Tree-based Multicast Transfer Protocol (TMTP) uses a clustering of nodes to address the case of a single sender delivering information to a large receiver set [YAV95]. Receivers are organized into a hierarchy of groups, which roughly correspond to the nodes' physical layout in the network. TMTP builds a tree control topology using the hop counts (number of intermediate steps over routers) between the receivers in the multicast group, by selecting the closest retransmitter for each receiver based on its hop count. For each of the groups of nodes constructed by this method, a *domain manager* is elected. This domain manager has the responsibility of ensuring successful delivery to all of the nodes within its group. NACKs are broadcast to the multicast channel, but their scope is limited by their time-to-live (TTL) field. The TTL field of a packet limits the range of a multicast packet as it propagates over multicast routers. In this way, multicast control messages from one domain do not reach all other domains, and so data repair operations are localized.

The Single Connection Emulation (SCE) architecture is designed to provide a link between the unicast transport layer and the multicast network layer [TAL95]. SCE provides an interface for a reliable multicast application to treat a receiver set as a single destination, by redefining unicast transmission terminology so that it applies to multicast groups. For example, the unicast function of establishing a connection is replaced by the multicast function of at least *n* receivers connecting to the SCE group. SCE was implemented using TCP connections to the receiver set, so it is subject to the ACK implosion scalability problem.

For problems of bulk data distribution from a single source, dividing the receiver set can be advantageous [AMM92]. By partitioning the receiver set into subgroups based on delay characteristics and throughput achieved from the source, the sender is not rate-limited by the slowest receiver. Group members with high throughput are grouped together and receive the data in less time than slow subgroups.

The main limitation of sender-initiated protocols is that the sender must have complete knowledge of the receiver set in order to ensure delivery, and the sender must keep state information for the entire group [LEV96A]. In very large multicast groups, complete group information is impossible to obtain, and therefore sender-initiated protocols cannot reach high levels of scalability.

2.3 Receiver-Initiated Approaches

Receiver-initiated protocols eliminate the requirement for the sender to have total information on the entire receiver set, and thus they can provide improved scalability. However, ensuring reliability becomes a more difficult problem. It may be argued that without explicit knowledge of the entire receiver set, the sender can never truly guarantee reliable transmission to every member of the group. However, in practice a weaker definition of reliability from the receiver's point of view can suffice.

The Scalable Reliable Multicast (SRM) protocol is a successful implementation of a receiver-based protocol [FLO95]. Rather than using a clustering scheme to organize nodes, SRM makes use of a homogenous receiver set where all nodes multicast their NACKs to the group and all nodes are capable of retransmitting data if it is available. To avoid implosion, SRM uses NACK suppression via slotting and damping much like XTP. The distribution of the random amount of introduced delay time for slotting is based upon a heuristic of the group's inter-node network latencies. Also, when a node wishes to respond to a NACK by retransmitting data, it slots and damps so that redundant data retransmissions do not overwhelm the multicast channel either.

While this approach scales well to multicast groups of up to a hundred users [FLO95], the heuristic delays necessary to prevent NACK and retransmission implosion grow with the size of the group, resulting in poor scalability. Large delays result in large NACK and retransmission latencies, thereby slowing throughput. The NACK suppression algorithm used requires that every node maintains timers based on updates multicast by every other node. As the group size gets larger, nodes must each do an increasing amount of work to maintain these timers [LEV96A]. The session messages in SRM used to calculate its heuristic measures of network latencies can actually use up more bandwidth than the application's useful data [HAN98]. This performance dependency on group size prevents high levels of scalability.

The scalability of SRM can be improved by limiting the scope of error recovery traffic, so that retransmissions are sent to a subset of the whole group rather than the entire receiver set. Splitting the destination set can be done by using multiple multicast groups for packet retransmission [KAS96]. By using multiple multicast groups for packet retransmission, receivers respond to packet loss by dynamically joining and leaving separate multicast groups where retransmissions are broadcast. This process limits the set of receivers who receive a retransmission, as retransmissions are only distributed to the set of receivers that joined the specific retransmission's multicast group [KAS96].

Other approaches to splitting the receiver set are hop-based scope control and local recovery groups, which both offer scalability improvements compared to SRM [LIU97]. Hop-based scope control uses the TTL field of the multicast packet to limit the reach of retransmission requests and retransmitted data. This ensures that data repair operations localized in one part of the multicast group do not affect other nodes in the group. Local recovery groups are separate multicast groups built from nodes that are close to each other in the multicast distribution tree. These multicast groups are used for localized data repair operations.

In addition, work has been done to improve the efficiency of the timers used for slotting and damping in SRM [GRO97] [NON98]. The Deterministic Timeouts for Reliable Multicast (DTRM) algorithm provides a method of computing optimal deterministic timeout values for each receiver in the multicast distribution tree, given the distribution tree topology and the sender-to-receiver round-trip delays [GRO97]. These timeout values can be used to slot and damp control messages, avoiding implosion. Improvements have also been made by using exponentially distributed timers, where probabilistic feedback is based on round-trip delays [NON98]. This method achieves scalability by providing low NACK latencies combined with good NACK suppression performance.

Another approach to solve the performance dependency on group size is to divide the receiver set into distinct groups in order to distribute the responsibility of handling retransmission requests.

The Reliable Multicast Transfer Protocol (RMTP) is a receiver-initiated extension of the tree structure used in the sender-initiated TMTP protocol [LIN96]. Regions are defined corresponding to groups of nodes in physical proximity over the network. Rather than using hop counts as the basis for defining regions as TMTP does, RMTP uses propagation delay as its measure. For each region a *designated receiver* is chosen. The designated receivers aggregate the control messages within their region and forward them to the sender. Since only the designated receivers send messages to the sender, implosion problems are eliminated.

The Tree-based Reliable Multicast Protocol (TRAM) is another receiver-initiated protocol based on a tree-based control topology [CHI98]. *Repair heads* are designated as being responsible for handling retransmission requests within tree groups. The TRAM tree management allows for the tree to dynamically change based upon feedback from the receiver set. This feedback consists of control messages from each receiver to its repair head containing data such as transmission statistics, congestion condition reports, and the number of available repair heads on the tree. Receivers may switch to a different repair head in order to distribute the load more efficiently. This information is also aggregated upwards on the tree, so that control traffic to the sender is reduced. Based on this feedback, the sender can change its data rate to match network conditions.

One problem with many of the reliable multicast protocols discussed so far is that the protocols are not well suited for all applications. For example, some applications may impose complete ordering and reliability requirements on their data transmissions, whereas other applications may only desire basic UDP-type service. The Tunable Multicast Protocol (TMP) addresses this problem of different application requirements

[BAS97]. TMP's reliability mechanisms are based on those of SRM. TMP provides a tunable reliability space consisting of the following dimensions: ordering, reliability, group size, group membership, persistence, and receiver storage. An application can set parameters for each of the dimensions to tune the protocol to its specific needs. A key element of TMP is that it uses the concept of *logical persistence*, where the data kept at nodes for possible future retransmission is not based on the age of the data (temporal persistence) nor the size of the retransmission buffer (spatial persistence), but rather it is based on application-defined logical units.

The MESH framework provides a flexible structure for large-scale multicast transport [LUC98]. The name MESH is derived from its self-organizing, soft-state recovery structure. MESH uses domain-scoped multicast transmissions, which are supported in IPv6 multicasting. To distribute control processing across the multicast group, MESH partitions the multicast group into subgroups based on network domain boundaries, and organizes the subgroups into hierarchies. Example hierarchy levels are LAN segments, campus networks, and regional backbones. Group members within a domain elect an *active receiver* (AR) to aggregate and forward domain control information to the next domain in the hierarchy. As presented, MESH supports two variations: MESH-R for reliable data distribution, and MESH-M for deadline-driven quality of service requirements.

The STORM (STructure-Oriented Resilient Multicast) protocol also allows the application to specify its own reliability requirements [XU97]. STORM is designed to improve the perceived performance of conferencing applications. STORM allows for each receiver in a multicast group to make a tradeoff between reliability and latency in data delivery. The design of STORM assumes that some users in a conference may be passive, i.e. they do not require much interactivity with the sender. Such users who do not require a high degree of interactivity can maintain a larger buffer of received data that has not yet been presented to the user. This larger buffer allows retransmissions time to occur before the real-time data is used, thereby providing a higher quality transmission for the user. However, the larger buffer size also means a higher latency between when the sender transmits data and when the data is presented to the user. STORM builds a logical structure for error recovery that consists entirely of application endpoints, so it does not incorporate features within the multicast distribution tree. This structure is a multi-parent tree where each node keeps a list of parent nodes to which it balances retransmission requests.

The Reliable Multicast Architecture for the Internet (IRMA) makes use of reliable unicast TCP connections at the end hosts in its control topology to efficiently ensure reliability [LEE99]. Each host can use the standard TCP/IP protocol stack in its transmissions, essentially treating the communication as a unicast connection. IRMA introduces special reliability functionality in a subset of multicast routers to create a virtual network, thus providing a reliable multicast framework that the TCP-based hosts can tap into. However, the success of this protocol is somewhat dependent on widespread deployment of the IRMA architecture.

Other protocols make use of extensions to the network layer in order to get performance gains. The On-Tree Efficient Recovery using Subcasting (OTERS) protocol uses multicast route backtracing and subcasting extensions to the network layer to achieve better performance [LI98]. Backtracing is a facility by which a member of a multicast group can determine the sequence of multicast-capable routers used between itself and a node in the group sending data. Subcasting is a facility to multicast a packet over a subtree of the multicast delivery tree, specified by the multicast group and the multicast-capable router at the root of the subtree. These extensions allow OTERS to build control topologies which match more closely with the underlying physical network structure. In addition, the subcasting extension lets OTERS perform transmission to a specified subset of the multicast delivery tree, hence conserving bandwidth. OTERS has been shown to have performance benefits over SRM and TMTP.

In the same vein, another reliable multicast protocol called Tracer makes use of MTRACE packets in IGMP to organize receivers of a multicast group deterministically into a logical tree structure [LEV98]. Tracer allows a receiver host to trace its path through the multicast group back to the source. This information can then be used to organize local error recovery schemes. Tracer has an emphasis on *packet-loss correlation*, meaning that parent-child relationships in the control topology also relate to how packets propagate through the multicast delivery tree. This minimizes the amount of redundancy in retransmission requests. Tracer can also make use of router improvements such as the subcasting extension used in OTERS.

A tree-based control topology is currently the predominant design of truly scalable reliable multicasting protocols, and protocols that do not make use of a tree or tree-like structure suffer from poor scalability [LEV96A]. However, trees are optimized for application domains that have one sender sending data to many receivers. Such protocols can suffer from performance penalties in cases where there are multiple senders of information [LIE98B]. Thus no existing protocol can offer both scalability to large groups of users and the ability to handle multiple data sources efficiently. This research addresses the limitations of tree-based protocols, with the goal of finding a solution that builds upon the tree-based control topology and improves its performance for multiple data sources.

3 The Hypercube Approach

Work on the hypercube structure done by J. Liebeherr and B. S. Sethi established the theoretical underpinnings of the hypercube control topology [LIE98A]. Nodes in a multicast group are arranged as vertices on a logical hypercube. This work provided an algorithm that generates an embedded tree rooted at any node within an incomplete hypercube. Embedded trees rooted at nodes sending data can then be used to aggregate control messages for scalable reliable multicast service. Liebeherr and Sethi showed the theoretical benefits of using a hypercube as opposed to a shared tree in a multicast group with multiple senders.

3.1 Hypercube Structure

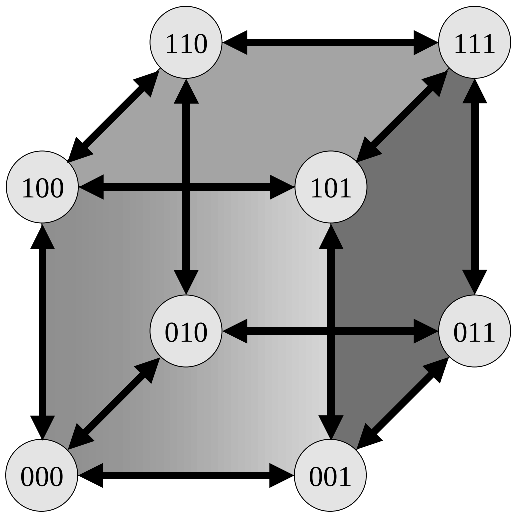


Figure 9. Hypercube nodes and their bit string labels.

An *n-dimensional hypercube* is a graph with $N = 2^n$ nodes. Each node is labeled by a bit string $k_n \ldots k_1$, where $k_i \in \{0, 1\}$. Nodes in a hypercube are connected by an edge if and only if their bit strings differ in exactly one position. A hypercube of dimension $n = 3$ is shown in Figure 9.

Previous literature in the field of parallel computing has produced algorithms for embedding spanning trees in a hypercube [LAK90] [LEI92] [QUI94], i.e. they create trees for a node set V which contain all nodes in V. However, these algorithms primarily work with static hypercubes, where the hypercube membership is known and immutable. Additionally, most algorithms assume a *complete* hypercube, where the number of nodes is equal to a power of two. However, multicast applications cannot use the same assumptions of static membership and hypercube completeness since group members may join and leave at any time.

The problem that must be addressed in particular is how to embed spanning trees in *incomplete* hypercubes, where the number of nodes is not a power of two. With an incomplete hypercube, it is required that the embedded spanning trees with node set V only contain nodes in V [LIE98B]. Trees that satisfy this property are *completely contained* within the incomplete hypercube. Complete containment is necessary so that embedded trees do not reference nodes which are not present in the incomplete hypercube.

One basic requirement in order to embed spanning trees in an incomplete hypercube is that the incomplete hypercube must not be disjoint, i.e. there must be a path from any node in the incomplete hypercube to any other node. A constraint that ensures that a hypercube is not disjoint is *compactness*: the dimension of the hypercube must be as small as possible. In other words, in an incomplete hypercube of N nodes the dimension of the hypercube must be the smallest integer not less than $\log_2(N)$.

To determine if an incomplete hypercube satisfies the compactness assumption, an ordering of the hypercube nodes' bit string labels is needed. The ordering is used to relate the bit string labels to the compactness property. A simple ordering based on the binary representation of the node label bit string is as follows [LIE98B]: Let $A = a_n a_{n-1} \ldots a_2 a_1$. Then $Bin(A) = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$. *Bin* is therefore an ordering based simply on the binary value of the label's bit string.

However, the binary form of the bit string itself is difficult to use in hypercube operations. For example, the nodes with labels "011" and "100" have consecutive binary labels, however they are quite distant in the hypercube as they differ in three bit positions (Figure 9). The result of this property is that the *Bin* operator is difficult to use for the task of creating spanning trees, since the binary value of a node's label does not readily indicate its position in the hypercube. A different way of ordering the labels is necessary, where the ordering can be efficiently related to the positions of the nodes in the hypercube.

This ordering can be accomplished by using a *Gray code*, where the Gray code is denoted by the operator $G()$. A Gray code is defined by the following three properties [QUI94]:

1. The values are unique. That is, if $G(i) = G(j)$, then $i = j$.
2. $G(i)$ and $G(i + 1)$ differ in only one bit, for $0 \leq i < 2^{d-1} - 1$.
3. $G(2^{d-1} - 1)$ and $G(0)$ differ in only one bit.

It can be shown that a Gray code for a number i is the bitwise exclusive-or of $Bin(i)$ right-shifted by one bit and $Bin(i)$ itself. That is,

$$G(i) := Bin(i) \text{ XOR } Bin(i / 2),$$

where “*i* / 2” represents integer division and XOR is the exclusive-or operator.

Table 1: Example Gray codes.

Index *i*	0	1	2	3	4	5	6	7
Bin(*i*)	000	001	010	011	100	101	110	111
G(*i*)	000	001	011	010	110	111	101	100

Example Gray codes $G(0)$ through $G(7)$ are listed in Table 1 above.

From the definition of a Gray code, the progression of Gray codes for consecutive indices has the property that each successive code differs from its predecessor by exactly one bit. Recall that neighboring nodes in a hypercube have bit string labels that differ in exactly one bit. Thus the Gray code establishes a complete ordering of all node labels, where successive Gray codes correspond to neighbors in the hypercube.

The Gray code provides a tool that can be used to relate the bit string labels of a hypercube’s nodes to its compactness. A hypercube of N nodes is compact if the bit string labels of the nodes are equal to $G(0)$ through $G(N - 1)$.

3.2 Embedding Trees

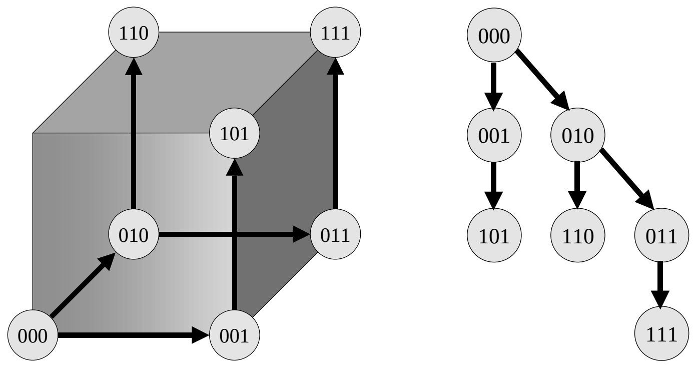
To create an embedded tree within a hypercube rooted at node R, it is sufficient to specify a method for each node to know its parent’s label. This task can be accomplished at each node given the knowledge of the node’s own bit string and the root node R. Since each node computes its link in the tree without aid from other nodes, the construction of the tree is distributed across the group membership.

Let $G^{-1}()$ be defined as the inverse Gray operator, such that $G^{-1}(G(i)) = i$. The following pseudocode algorithm from Liebeherr and Sethi’s work designates a node’s parent, based on the node’s own bit string label and the label of the tree’s root [LIE98B]:

```
Input:  Label of the i-th node in the Gray encoding:
            $G(i) := I = I_n \ldots I_2 I_1$,
            and the label of the r-th node ($\neq i$) in the Gray encoding:
            $G(r) := R = R_n \ldots R_2 R_1$.
Output:  Label of the parent node of node I in the embedded tree rooted at R.
Procedure Parent(I, R)
    If ($G^{-1}(I) < G^{-1}(R)$)
        // Flip the least significant bit where I and R differ.
        Parent := $I_n I_{n-1} \ldots I_{k+1}(1 - I_k) I_{k-1} \ldots I_2 I_1$
            with $k = \min_i(I_i \neq R_i)$
    Else // ($G^{-1}(I) > G^{-1}(R)$)
        // Flip the most significant bit where I and R differ.
        Parent := $I_n I_{n-1} \ldots I_{k+1}(1 - I_k) I_{k-1} \ldots I_2 I_1$
            with $k = \max_i(I_i \neq R_i)$
    EndIf
End
```

Figure 10: Tree Embedding Algorithm.

The use of this algorithm to embed a spanning tree in an incomplete hypercube is illustrated in the following examples from Liebeherr and Sethi's work [LIE98B]:



a) Embedded in hypercube **b) Resulting tree**

Figure 11: Embedded Tree with "000" as Root.

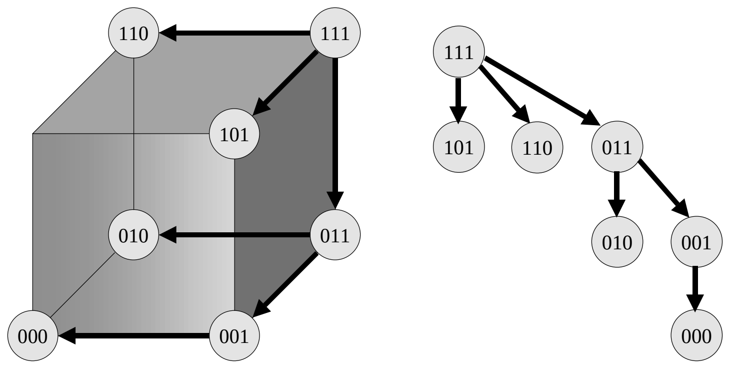


Figure 12: Embedded Tree with "111" as Root.

In Figure 11, a tree rooted at node "000" is embedded within an incomplete hypercube using the algorithm described above. In Figure 12, the same incomplete hypercube is shown with a tree rooted at node "111". Note that the algorithm generates spanning trees that are wholly contained within the incomplete hypercube, so no link references a label that is not present within the incomplete hypercube.

3.3 Theoretical Benefits of the Hypercube

Using the aforementioned method for embedding spanning trees within an incomplete hypercube, Liebeherr and Sethi then were able to rigorously compare the theoretical performance of the hypercube control topology with a tree topology [LIE98B].

Recall from Chapter 1 that a *K*-ary shared tree is a tree where each node has at most *K* children, and it is re-hung as needed so that nodes sending data are at the root of the tree [LEV96B]. For both the hypercube and the *K*-ary shared tree, spanning trees rooted at the sender are used for aggregation of control information. For the hypercube to have better theoretical performance than the *K*-ary shared tree for multiple senders of data, it must be shown that the spanning trees rooted at all nodes of the hypercube have better performance characteristics on average than the spanning trees rooted at all nodes of the *K*-ary shared tree.

A spanning tree over node set *V* rooted at node $l \in V$ is denoted as T_l. The spanning trees T_l are constructed for the hypercube and the *K*-ary shared tree as follows:

- **Hypercube**: Using the algorithm presented in Figure 10, the tree T_l with node *l* as root is embedded in a hypercube of *N* nodes.
- ***K*-ary Shared Tree**: An initial balanced *K*-ary tree of *N* nodes rooted at a fixed node *r* is re-hung with node *l* as the root to form T_l.

Liebeherr and Sethi compared the attributes of *K*-ary shared tree and hypercube control topologies in terms of several measures [LIE98B]:

- The number of children at a node *k* in tree T_l, $w_k(T_l)$

  The number of children of each node *w* relates to the network load distribution across the tree. Since each child node can send control messages to its parent, a node with *w* children receives a number of control messages proportional to *w*. If *w* is large, the node suffers from a bottleneck due to the high load of processing many control messages.

- The number of descendants in the sub-tree below a node *k* in tree T_l, $v_k(T_l)$

  The amount of state information that a protocol maintains for reliability may also be related to the number of descendants of a node, *v*. Since each node is responsible for ensuring the reliability of all the nodes below it in the tree, some reliable protocols may have scalability characteristics that depend on *v*.

- The path length from a node *k* in tree T_l to the root of the tree, $p_k(T_l)$

The path length *p* affects total end-to-end latency. In the worst case, a control message must propagate from node of the tree all the way back to the root before a retransmission can be issued, and the total number of these steps is the path length. Each step from one node to another along the way increases the total delay time from the detection of a lost packet to the lost packet's retransmission. A low path length results in higher theoretical maximum data rates, whereas a high path length results in low bandwidth.

These characteristics are measured at each node *k* in a particular rooted spanning tree T_l. These measures are condensed by taking the average over all *N* possible root nodes *l* for each of the *K*-ary shared tree and the hypercube:

$$w_k := \frac{1}{N}\sum_{l=1}^{N} w_k(T_l)$$
$$v_k := \frac{1}{N}\sum_{l=1}^{N} v_k(T_l)$$
$$p_k := \frac{1}{N}\sum_{l=1}^{N} p_k(T_l)$$

Both the average and maximum values of these measures at each node can be computed:

$$w_{avg} := \frac{1}{N}\sum_{k=1}^{N} w_k \qquad v_{avg} := \frac{1}{N}\sum_{k=1}^{N} v_k \qquad p_{avg} := \frac{1}{N}\sum_{k=1}^{N} p_k$$
$$w_{\max} := \max_k w_k \qquad v_{\max} := \max_k v_k \qquad p_{\max} := \max_k p_k$$

For the rooted spanning trees of a hypercube of *N* nodes, these measures are summarized as follows [LIE98B]:

Table 2: Theoretical measures of hypercube topology.

$$w_{\text{avg}} = 1 - \frac{1}{N}$$

$$w_{\text{max}} = 2 - \frac{\log_2 N + 2}{N}$$

$$v_{\text{avg}} = \frac{1}{2}\log_2 N + 1$$

$$v_{\text{max}} = \frac{1}{8}(\log_2 N)^2 + \frac{3}{8}\log_2 N + 1$$

$$p_{\text{avg}} = \frac{1}{2}\log_2 N$$

$$p_{\text{max}} = \frac{1}{2}\log_2 N$$

For the rooted spanning trees of a shared *K*-ary tree of *N* nodes and depth *d*, these measures are summarized as follows [LIE98B]:

Table 3: Theoretical measures of *K*-ary tree.

$$w_{\text{avg}} = 1 - \frac{1}{N}$$

$$w_{\text{max}} = K - \frac{1}{N}$$

$$v_{\text{avg}} = 2d + \frac{K-5}{K-1} + \frac{6d}{N(K-1)} + \frac{4(K-2)}{N(K-1)^2} + \frac{4(d+1)}{N^2(K-1)^2}$$

$$v_{\text{max}} = \begin{cases} \dfrac{5N}{8} + \dfrac{1}{4} - \dfrac{11}{8N} & \text{if K = 2} \\ \dfrac{(K-1)N}{K} + \dfrac{2}{K} - \dfrac{1}{KN} & \text{otherwise} \end{cases}$$

$$p_{\text{avg}} = 2d - \frac{4}{K-1} + \frac{6d}{N(K-1)} + \frac{4(K-2)}{N(K-1)^2} + \frac{4(d+1)}{N^2(K-1)^2}$$

$$p_{\text{max}} = 2d - \frac{3}{K-1} + \frac{3(d+1)}{N(K-1)}$$

From these results, the benefits of using a hypercube as the control topology can be directly analyzed. Note that many of the performance characteristics of the *K*-ary tree are directly proportional to *d*, *K*, or *N*, whereas the hypercube performance characteristics are of order $O(\log_2 N)$. This shows that as the multicast group size increases, the hypercube's load factors grow at a slower rate than that of the K-ary tree.

Using empirical data, Liebeherr and Sethi also showed that in real-world group memberships the hypercube offers significant performance advantages compared to other control topologies [LIE98B].

4 The HyperCast Protocol

In this chapter the details of the HyperCast protocol are presented. The HyperCast protocol provides for the scalable construction and maintenance of a hypercube structure suitable for use with the tree embedding algorithm discussed in Chapter 3.

The data structures, protocol messages, node states and state transition mechanisms used in the design of the protocol are listed in full. Examples of the protocol in action are also given, as well as details of the protocol implementation.

4.1 Overview

The goal of the HyperCast protocol is to maintain the logical hypercube structure so that reliability mechanisms can be easily overlaid on top of it, regardless of how nodes join or leave the structure. It is assumed that the sole goal of the protocol is to maintain the hypercube structure so that trees can be embedded within it as discussed in Chapter 3, since actual data transmission and error correction mechanisms can be implemented separately. Since embedded trees are created within the hypercube structure, existing reliability schemes designed for tree structures can be ported to use the embedded trees within the hypercube.

All nodes wishing to participate in the hypercube structure join a single multicast group, referred to as the *control channel*. Every node can both send and receive messages on this control channel. Only HyperCast control messages are distributed using this channel; data and repair transmissions are distributed separately.

The protocol is *soft-state*, meaning that the state information kept at nodes is periodically refreshed by HyperCast messages without requiring a consistent state at all times. State information that is not refreshed will expire. This design feature allows for the protocol to be tolerant of network delays and packet losses. Each node has information only about its neighbors in the hypercube, and no entity in the system has complete information about the whole group membership.

Nodes in the hypercube each have an associated *physical address*, given by the pair of their IP address and the UDP port being used for the HyperCast protocol. Due to this representation, all nodes are guaranteed to have distinct physical addresses.

In addition each node has a *logical address*, given by the bit string label discussed in Chapter 3. For an N-dimensional hypercube, N bits are needed in the logical address to give a unique logical address to each node. In the HyperCast protocol, logical addresses are represented as 32-bit integers, with one bit reserved to designate an invalid logical address. Therefore the protocol allows for hypercubes of up to 2^{31} (approximately two billion) nodes.

The hypercube is in a *stable* state if it satisfies the following three criteria:

1. ***Consistent***: No two nodes share the same logical address.
2. ***Compact***: In a multicast group with N nodes, the nodes have bit string labels equal to $G(0)$ through $G(N-1)$.
3. ***Connected***: All nodes know the physical address of each of their neighbors in the hypercube.

Nodes joining the hypercube, nodes leaving the hypercube, and network faults can cause a hypercube to violate one or more of the above conditions, leading to an *unstable* state. The task of the HyperCast protocol is to continuously return the hypercube to a stable state in a reliable and efficient manner.

4.2 Basic Data Structures

A *neighbor* of a node *A* is another node *B* linked to node *A* via an edge of the hypercube. As discussed in Chapter 3, hypercubes have the property that each node's logical address differs from each of its neighbors' logical addresses by exactly one bit. This property is useful for the protocol, since every node can determine the logical addresses of all of its potential neighbors based only on its own logical address. To determine in an incomplete hypercube which neighbors should be present, a node also requires knowledge of the highest logical address in the hypercube (unless otherwise stated, it is assumed that the ordering of logical addresses will be based on the $G^{-1}()$ operator applied to the node labels). For example, if a node with logical address "001" knows that the highest logical address in the hypercube is "010", then it will not expect a neighbor to be present with logical address "101", since $G^{-1}(101) > G^{-1}(010)$.

Every node keeps data about its neighbors in a *neighborhood table*. Every potential neighbor of the node has an entry in the table, consisting of the following data:

- The neighbor's logical address
- The neighbor's physical address, if known
- The time elapsed since the node last received a message from the neighbor
- The time elapsed since the node began attempts to contact its neighbor

In addition, every node keeps track of the current highest logical address in the hypercube, so that it can determine which of its neighbors should be present in its neighborhood table. As with entries in the neighborhood table, the node keeps a record of information for the highest logical address, consisting of the following data:

- The logical address of the highest known node in the hypercube

- The time elapsed since the node last received a message from the node with the highest logical address
- The last received sequence number from the node with the highest logical address

The node with the highest logical address attaches sequence numbers to the multicast messages it sends, as will be discussed in Section 4.5. Nodes store this sequence number so that they can determine if they have received recent or outdated information.

4.3 HyperCast Timers and Periodic Operations

Four time parameters are used in the HyperCast protocol. These parameters and their uses are defined below and listed with their values used in this implementation:

- $t_{\text{heartbeat}}$ (2s): Nodes send messages to each of their neighbors in the neighborhood table periodically at intervals separated by the time $t_{\text{heartbeat}}$.
- t_{timeout} (10s): When the time elapsed since a node last received a message from a neighbor becomes greater than the time t_{timeout}, the neighbor's entry is said to be *stale* and the neighborhood table is *incomplete*. A missing neighbor is referred to as a *tear* in the hypercube. In addition to the neighborhood table entries, the information about the highest known node in the hypercube also becomes stale after a period of time t_{timeout}.
- t_{missing} (20s): As will be discussed in Section 4.5, after a neighbor's entry becomes stale the node then begins multicasting on the control channel to contact the missing neighbor. If the missing neighbor fails to respond during another period of time t_{missing}, the node removes the missing neighbor's entry

from the neighborhood table and proceeds under the assumption that the neighbor has failed.

- t_{joining} (6s): Nodes that are in the process of joining the hypercube send multicast messages to broadcast their presence to the entire group, as will be discussed in Section 4.5. To prevent a large number of joining nodes from saturating the control channel with multicast messages, a joining node that receives a multicast message from another joining node will suppress its own message and wait for a period of time t_{joining} before attempting to broadcast its message.

4.4 Node States

In the HyperCast protocol, each node in the hypercube is in one of eleven different states. Based on events that occur and HyperCast control messages that are received, nodes transition between states.

Table 4: Node state definitions.

Outside:	Not yet participating in the group.
Joining:	Wishes to join the hypercube, but does not yet have any information about the rest of the hypercube. Its logical address is marked as invalid.
JoiningWait:	A **Joining** node that has received a *beacon* from another **Joining** node within the last t_{joining} time
StartHypercube:	Has determined that it is the only node in the multicast group since it has not received any control messages for a period of time t_{timeout}, and it starts its own stable hypercube of size one.
Stable:	Knows all of its neighbors' physical addresses.
Incomplete:	Does not know one or more of its neighbors' physical addresses, or a neighbor is assumed to have left the hypercube after not receiving pings from that neighbor for a period of time t_{timeout}.
Repair:	Has been **Incomplete** for a period of time t_{missing} and it begins to take actions to attempt to repair its neighborhood.
HRoot/Stable:	**Stable** node which also believes that it has the highest logical address in the entire hypercube, as ordered by the $G^{-1}()$ operator.
HRoot/Incomplete:	**Incomplete** node which also believes that it has the highest logical address in the entire hypercube, as ordered by the $G^{-1}()$ operator.
HRoot/Repair:	**Repair** node which also believes that it has the highest logical address in the entire hypercube, as ordered by the $G^{-1}()$ operator.
Leaving:	Node that wishes to leave the hypercube.

Note that a hypercube of N nodes is in a stable state if all of its nodes have unique logical addresses from $G(0)$ to $G(N\text{-}1)$ and are in the **Stable** state, with the exception of the node with a logical address equal to $G(N\text{-}1)$ which is in the **HRoot** state.

4.5 Message Types

The functions of the hypercube are based upon a set of basic message types. There are a total of four message types that are used by the protocol. The protocol does not depend on the reliability of basic message transmission, since the soft-state design of the protocol allows for packet losses. Messages are sent via either a unicast transmission directly to another node, or a multicast transmission to the group's control channel.

Beacon Message: The *beacon* message is a message that is multicast on the control channel. A *beacon* contains the logical/physical address pair of the sender, as well as the logical address and sequence number of the currently known **HRoot**. There are three cases in which nodes periodically broadcast a *beacon* message at intervals spaced $t_{\text{heartbeat}}$ apart: (1) if the node considers itself to be the **HRoot**, (2) if the node determines that it has an incomplete neighborhood, or (3) if the node is in the process of joining the hypercube. The **HRoot** sends *beacons* to the whole group so that all nodes know the highest logical address in the hypercube, and therefore know which of their neighbors should be present in their neighborhood tables. The **HRoot** also adds a sequence number which it increments every time it sends a *beacon*. The sequence number is used to mark the timeliness of the information. Beacon messages with higher sequence numbers have more current information. A node with an incomplete neighborhood sends *beacons* periodically so that its missing neighbors (if present in the hypercube) are informed of the node's physical address and the nodes can reestablish their logical connection. **Joining** nodes periodically send *beacons* to advertise their presence to the group.

Ping Message: Each node periodically sends a *ping* to all of its neighbors listed in its neighborhood table to inform the neighbors that the node is still present in the hypercube. A *ping* is a short unicast message, containing the logical/physical address pair of both the sender and the receiver of the message, as well as the logical address and sequence number of the currently known **HRoot**. If a node has not received a *ping* from a neighbor for an extended period of time t_{timeout}, the node will consider its neighborhood incomplete and will begin sending *beacons* as described above. If it still has not received a *ping* from its neighbor after another period of time t_{missing}, it will assume that its neighbor has failed and will remove it from its neighborhood list. *Ping* messages are also used as a vehicle to assign a new logical address to the receiver of the *ping* message.

Leave Message: When a node wishes to leave the hypercube, it sends a *leave* message. Nodes receiving this *leave* message will remove the leaving node from their neighborhood tables.

Kill Message: Nodes receiving a *kill* message will immediately send *leave* messages to all of the neighbors in their neighborhood tables, and then enter the **Joining** state. The *kill* message is used for the elimination of nodes with duplicate logical addresses.

4.6 Message Packet Format

Basic messages are sent using the following packet format, common to all messages:

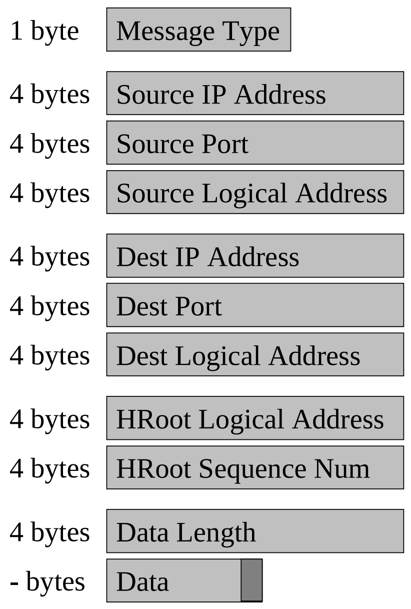


Figure 13: Packet format.

The **Message Type** field is defined as the following:

Table 5: Message Types.

Message Type:	*Ping*	*Beacon*	*Leave*	*Kill*
Field Value:	0	1	2	3

The **IP Address** fields are filled in the network address' most significant byte to least significant byte order. The **Port**, **Logical Address**, **Sequence Number**, and **Data Length** fields are also filled in most significant byte to least significant byte order.

Data is a variable-length field, with its length specified by the **Data Length** field of the packet. The **Data** field is present for future expansion, and it is not currently used in the protocol.

4.7 Protocol Mechanisms

The goal of the HyperCast protocol is to maintain a stable hypercube; i.e. the hypercube must be consistent, compact, and connected.

To maintain the consistency criterion, a mechanism called ***Duel*** (duplicate elimination) is employed. The ***Duel*** mechanism deterministically ensures that logical address conflicts are resolved by always eliminating the conflicting node with the lower physical address. If a node detects that another node has the same logical address, it compares its own physical address with the physical address of the conflicting node. Ordering of physical addresses is given by the 32-bit integer representation of the nodes' IP addresses. If two nodes share the same IP address, then the HyperCast control channel port number is used for comparison. If the node's physical address is numerically greater than the conflicting node's physical address using this ordering, the node with the greater physical address issues a *kill* message to the other node. Otherwise, it sends *leave* messages to all of its neighbors and rejoins the hypercube in the initial **Joining** state.

To maintain the compactness criterion, the ***Admin*** (address minimization) mechanism is used. The ***Admin*** function of the protocol continually attempts to move nodes into lower logical addresses whenever opportunities arise. When a node receives *beacon* messages from the **HRoot** node, it checks to see if it has been missing a neighbor in its own neighborhood list with a lower logical address than that of the **HRoot** for longer than a period of time t_{missing}. If such a tear is found, the node sends a *ping* with the vacant lower logical address back to the **HRoot**. A node that receives a *ping* message with a destination logical address lower than its actual logical address will set its logical address to the value given in the *ping*. Thus as tears appear in the hypercube, nodes migrate down to lower logical addresses, eventually filling the tears. The continual movement of nodes towards lower logical addresses as tears appear ensures that the hypercube remains compact.

The ***Admin*** mechanism also governs the process of nodes joining the hypercube. Initially, the logical address of a **Joining** node is marked as an invalid address. Since the **Joining** node periodically sends *beacons* to announce its presence to the group, other nodes check to see if they can find a “lower”, valid logical address for the new node in the hypercube. If a tear in the hypercube is found, the **Joining** node is sent a *ping* to lower it to the vacant logical address. If there is no tear in the hypercube, in order to maintain compactness the **Joining** node is placed at the next higher logical address above the **HRoot**. The **HRoot** is always capable of adding a node to its neighborhood list as its successor in the Gray ordering, so it sends a *ping* to the **Joining** node containing that new logical address. Therefore nodes join a stable hypercube as new **HRoots**.

The connectedness criterion is maintained by the following process: whenever a node *A* receives a message from another node *B* with a logical address that designates it as a neighbor, the logical/physical address pair of node *B* is added into node *A*’s neighborhood table. Subsequent *pings* ensure that the link between neighbors remains. If a neighbor does not send *pings* for a period of time as described in Section 4.5, it is assumed that the neighbor has dropped out of the hypercube and its entry in the neighborhood table is removed. Action taken by the ***Admin*** mechanism then can repair the hole in the neighborhood table.

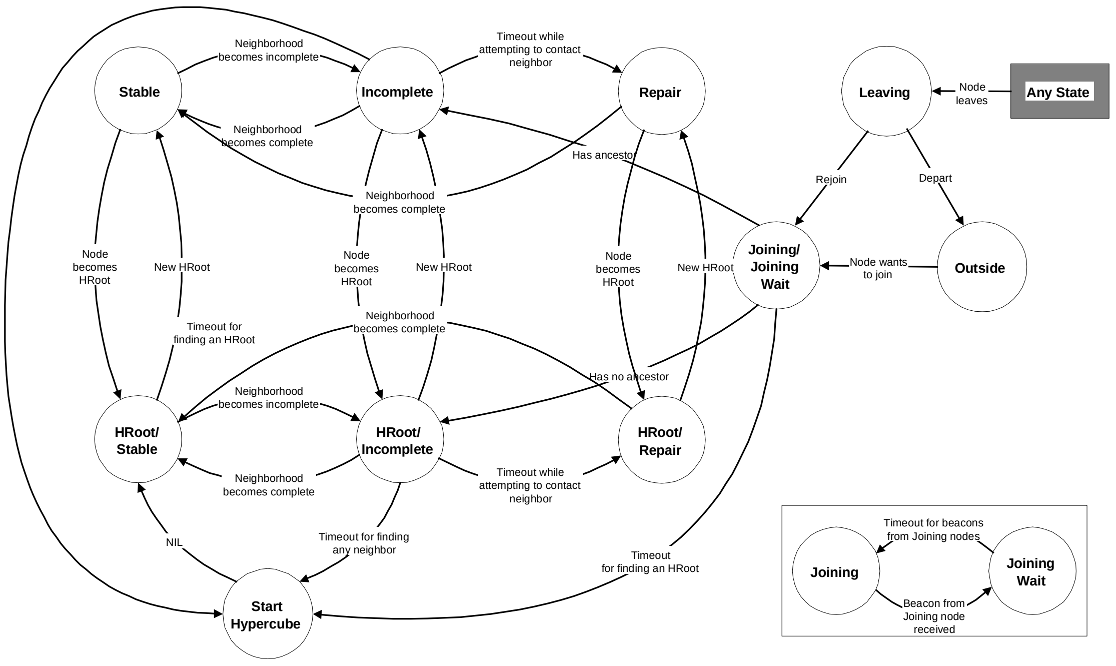

Stable
Incomplete
Repair
Leaving
Any State
Node leaves
Neighborhood becomes incomplete
Neighborhood becomes complete
Timeout while attempting to contact neighbor
Has ancestor
Rejoin
Depart
Joining/ Joining Wait
Outside
Node wants to join
Node becomes HRoot
New HRoot
Timeout for finding an HRoot
Has no ancestor
HRoot/ Stable
HRoot/ Incomplete
HRoot/ Repair
NIL
Timeout for finding any neighbor
Timeout for finding an HRoot
Start Hypercube
Timeout for beacons from Joining nodes
Joining
Joining Wait
Beacon from Joining node received

Figure 14: Node State Transition Diagram

Figure 14 summarizes the transitions between node states. Node states are represented as circles, and arrows between states denote the method of transitioning between two states.

4.8 Protocol Event Tables

The protocol actions taken by the nodes in response to events are presented in table form below. The "→" symbol means that the node will switch to the indicated state.

Table 6: Event table for Outside state.

Outside • Node is not part of the hypercube

Event:	**Action:**
Application wants to join HyperCast group	→ **Joining**

Table 7: Event table for Joining state.

Joining

- Wants to join the hypercube
- Logical address is set as *invalid*

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *beacon* message to control channel
No *ping* received for period t_{timeout}	→ **StartHypercube**
Beacon received from non-**Joining** node	Update known **HRoot** information
Beacon received from **Joining** node	→ **JoiningWait**
Ping received	Set own logical address to *ping*'s destination logical address
After *ping* received, own logical address equals known **HRoot**'s logical address	→ **HRoot/Incomplete**
After *ping* received, own logical address does not equal known **HRoot**'s logical address	→ **Incomplete**

Table 8: Event table for JoiningWait state.

JoiningWait

- Wants to join the hypercube
- Has received a *beacon* from a **Joining** node
- Logical address is set as *invalid*

Event:	**Action:**
No *ping* received for period t_{timeout}	→ **StartHypercube**
Beacon received from non-**Joining** node	Update known **HRoot** information
No *beacon* received from **Joining** node for period t_{joining}	→ **Joining**
Ping received	Set own logical address to *ping*'s destination logical address
After *ping* received, own logical address equals known **HRoot**'s logical address	→ **HRoot/Incomplete**
After *ping* received, own logical address does not equal known **HRoot**'s logical address	→ **Incomplete**

Table 9: Event table for StartHypercube state.

StartHypercube

- Start new hypercube

Event:	**Action:**
	Set own logical address to $G(0)$ → **HRoot**

Table 10: Common event table for several states.

Stable
Incomplete
Repair
HRoot/Stable
HRoot/Incomplete
HRoot/Repair

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *ping* message to all valid neighbors
Application triggers leave	Send *leave* message to all valid neighbors → **Leaving**
Receive message with source logical address equal to own logical address and source physical address is less than own	Send *kill* to message source
Receive message with source logical address equal to own logical address and source physical address is greater than own	Send *leave* to all valid neighbors → **Leaving**
Kill received	Verify that source's physical address is greater than own physical address Send *leave* to all valid neighbors → **Leaving**
Ping received	Update neighborhood entry for sender's logical address Update known **HRoot** information
Beacon received	Update known **HRoot** information
Leave received	Remove neighborhood entry for sender's logical address

Table 11: Event table for Stable state.

Stable

Event:	**Action:**
Neighborhood becomes incomplete due to lack of *pings* from a neighbor for period t_{timeout}	→ **Incomplete**
Own logical address is greater than known **HRoot** logical address	→ **HRoot/Stable**

Table 12: Event table for Incomplete state.

Incomplete

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *beacon* message to control channel
Neighborhood becomes complete	→ **Stable**
Own logical address is greater than known **HRoot** logical address	→ **HRoot/Incomplete**
Neighborhood partially empty for timeout interval t_{missing}	→ **Repair**
Neighborhood completely empty	→ **StartHypercube**

Table 13: Event table for Repair state.

Repair

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *beacon* message to control channel
Beacon received from **HRoot** or **Joining** node	Send *ping* message to *beacon* source containing new logical address to fill tear in neighborhood
Neighborhood becomes complete	→ **Stable**
Own logical address is greater than known **HRoot** logical address	→ **HRoot/Repair**
Neighborhood completely empty	→ **StartHypercube**

Table 14: Event table for HRoot/Stable state.

HRoot/Stable

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *beacon* message to control channel Increment sequence number
Beacon received from **Joining** node	Register **Joining** node as next higher neighbor Increment sequence number Update known **HRoot** information to be new **HRoot**
Neighborhood becomes due to lack of *pings* from a neighbor for period t_{timeout}	→ **HRoot/Incomplete**
Own logical address is less than known **HRoot** logical address	→ **Stable**

Table 15: Event table for HRoot/Incomplete state.

HRoot/Incomplete

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *beacon* message to control channel Increment sequence number
Beacon received from **Joining** node	Register **Joining** node as next higher neighbor Increment sequence number Update known **HRoot** information to be new **HRoot**
Neighborhood becomes complete	→ **HRoot/Stable**
Own logical address is less than known **HRoot** logical address	→ **Incomplete**
Neighborhood partially empty for timeout interval t_{missing}	→ **HRoot/Repair**
Neighborhood completely empty	→ **StartHypercube**

Table 16: Event table for HRoot/Repair state.

HRoot/Repair

Event:	**Action:**
Periodically, every $t_{\text{heartbeat}}$	Send *beacon* message to control channel Increment sequence number
Beacon received from **Joining** node	Send *ping* message to *beacon* source containing new logical address to fill tear in neighborhood
Neighborhood becomes complete	→ **HRoot/Stable**
Own logical address is less than known **HRoot** logical address	→ **Repair**
Neighborhood completely empty	→ **StartHypercube**

Table 17: Event table for Leaving state.

Leaving

- Waits for period t_{timeout} to ensure that neighbors receive *leave* messages in response to their *pings*
- Proceeds to **Outside** if leave was initiated by application, otherwise proceeds to **Joining**

Event:	**Action:**
Ping received	Send *leave* to message source
Leave was triggered by application and t_{timeout} time has elapsed	→ **Outside**
Leave was not triggered by application and t_{timeout} time has elapsed	→ **Joining**

4.9 Protocol Behavior Examples

It is advantageous to demonstrate the protocol's behavior by means of examples.

4.9.1 The Stable Hypercube

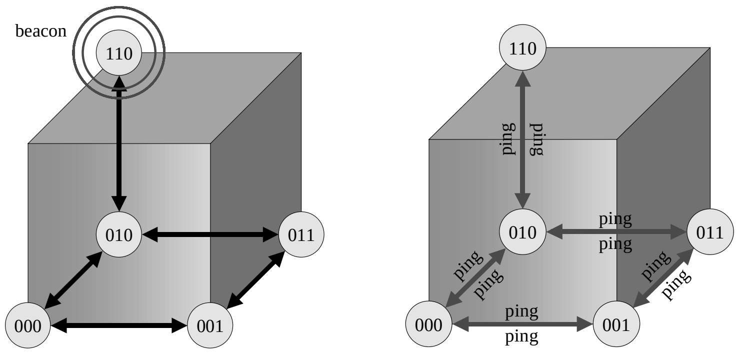


Figure 15: Stable hypercube.

In a stable hypercube, the **HRoot** multicasts *beacons* periodically (Figure 15-a). This keeps all nodes informed of the logical address of the **HRoot**, and therefore the nodes know which of their neighbors should be present in their neighborhood tables.

Every node also periodically sends *ping* messages to all the neighbors listed in its neighborhood table (Figure 15-b).

4.9.2 Adding a Node

The process of allowing nodes to merge into the hypercube control topology is shown here.

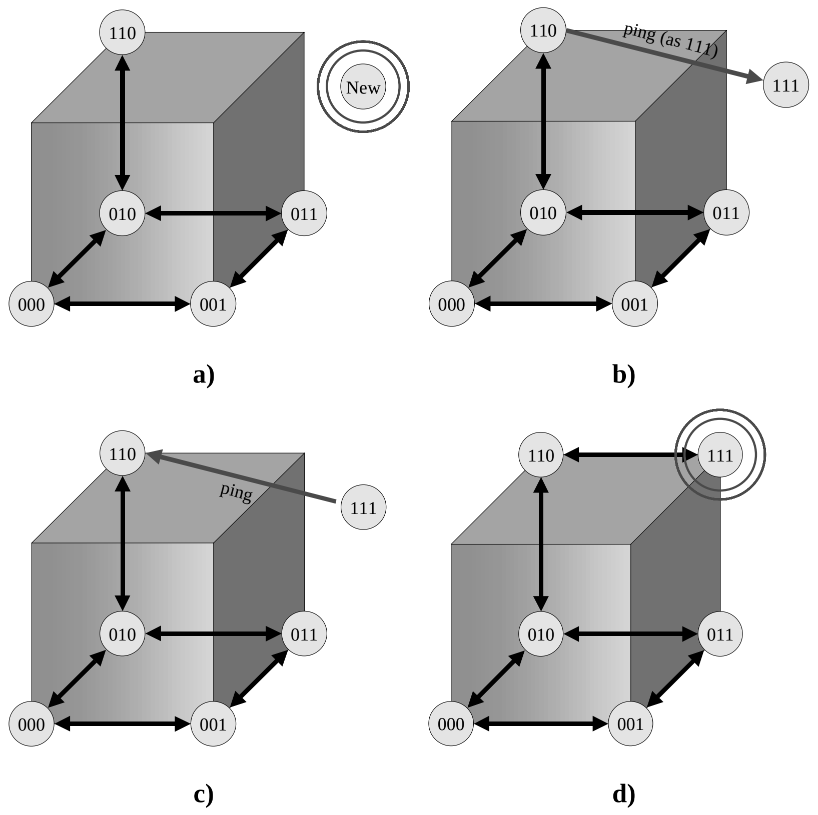


Figure 16: Joining node.

The **Joining** node periodically sends *beacon* messages, making its presence known to the group (Figure 16-a). The **HRoot** will place the **Joining** node as its neighbor at the next successive position in the hypercube, as ordered by the $G^{-1}()$ operator. The **HRoot** also knows that the new node will have the highest known logical address, so it updates its highest known logical address entry and enters the **Stable** state. It *pings* the new node with the new logical address (Figure 16-b). The new node takes on the new logical address and replies with a *ping* back to the original **HRoot** (Figure 16-c).

The new node determines from the *ping* packet that it is the **HRoot**, since its logical address is equal to the highest known logical address. It begins sending *beacons* as an **HRoot** (Figure 16-d).

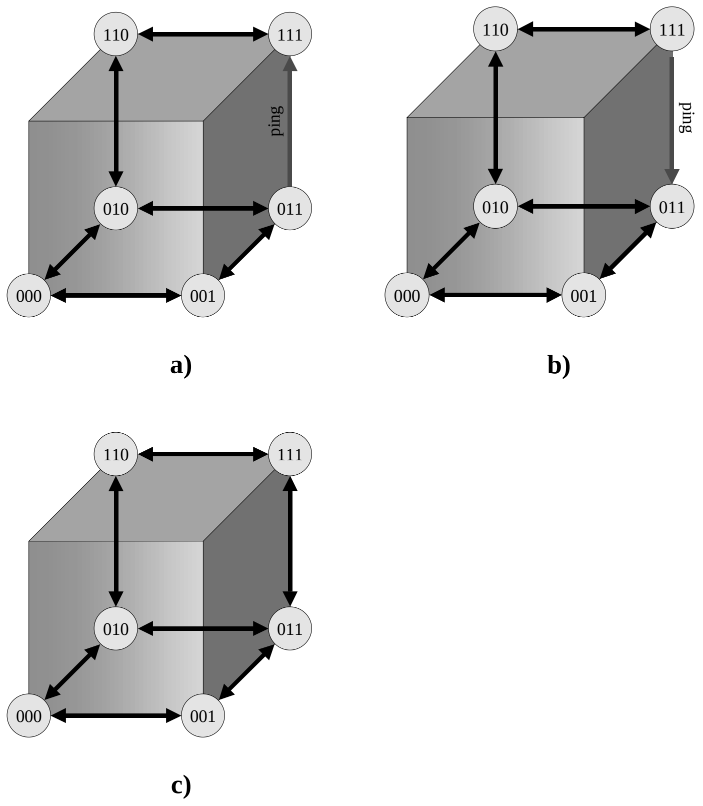


Figure 17: Joining node (continued).

Next, the new **HRoot**'s neighbors receive the *beacon* and reply with a *ping*, since the **HRoot** naturally belongs in their respective neighborhoods (Figure 17-a). The **HRoot** is then informed about its neighbors and replies with a *ping* (Figure 17-b). All nodes in the hypercube have complete neighborhood tables and know all their neighbors, so the hypercube is stable (Figure 17-c).

4.9.3 Repairing a Tear

The process of repairing defects in the hypercube control topology is shown here.

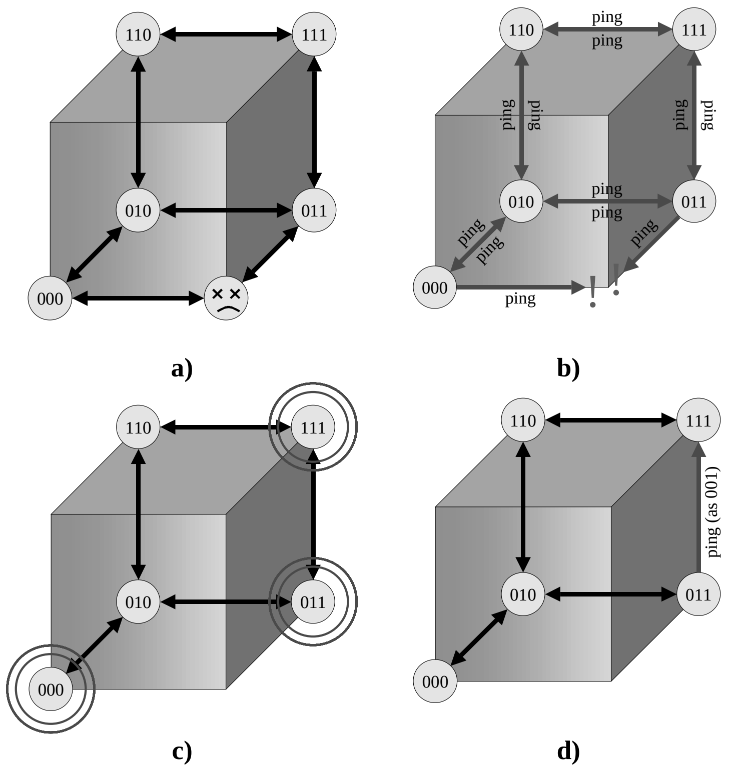


Figure 18: Repairing a tear.

It is possible that a node can fail unexpectedly (Figure 18-a). Nodes that have failed are detected because their neighbors do not receive *ping* messages from the failed node for a period of time t_{timeout} (Figure 18-b). Each of the failed node's neighbors then periodically sends *beacons* to indicate that they have detected a missing neighbor (Figure 18-c). Note that if the failed node returned at this time, the *beacons* from its neighbors will be used to reestablish the logical connections in its neighborhood table. After sending *beacons* for a period of time t_{missing} without receiving a reply, each neighbor assumes that the failed node will not return and a replacement is needed. The ***Admin*** mechanism then begins as one or more neighbors sends a *ping* to the **HRoot** to lower the **HRoot**'s logical address to fill the tear (Figure 18-d).

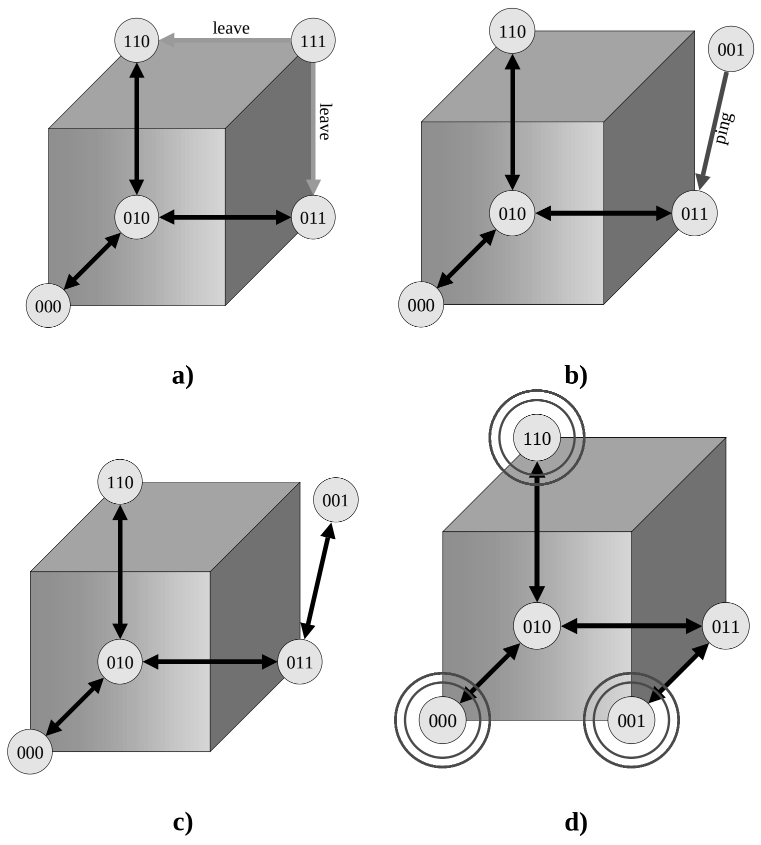
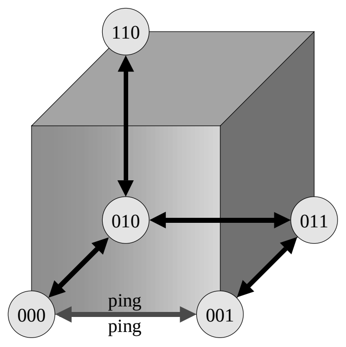
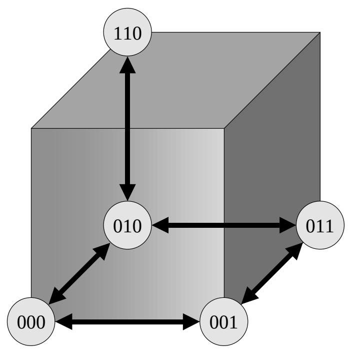


Figure 19: Repairing a tear (continued).

Upon receiving the *ping*, the **HRoot** sends *leave* messages to its neighbors to notify its neighbors that the **HRoot** will be leaving their neighborhoods (Figure 19-a). The **HRoot** then assumes the new logical address given to it by the failed node's neighbor, and replies to it with a *ping* of its own (Figure 19-b). This completes the logical connection between the two nodes, since both nodes have entries for each other in their respective neighborhood tables and know each other's physical addresses (Figure 19-c). The relocated old **HRoot** then *beacons*, since it does not yet know all of its neighbors (Figure 19-d).



a)



b)

Figure 20: Repairing a tear (continued).

The neighboring nodes receive each other's *beacons* and respond by sending *pings* (Figure 20-a). This completes the repair procedure, and the hypercube has returned to a stable state (Figure 20-b).

4.10 Implementation Details

The HyperCast protocol was implemented using the Java programming language. Java was chosen for its portability to multiple platforms and its easy-to-use threading constructs [CAM98]. Classes were written to encapsulate the physical address, logical address, and neighborhood table data structures. These classes contained functionality for basic operations, such as adding/removing a node's entry or searching for a tear in the neighborhood table. Additional classes were used to represent a HyperCast message and a queue used for storing HyperCast messages that have been received but not yet processed.

Two sockets were used for each hypercube node. A single datagram socket was used for unicast send and receive operations, and a single multicast datagram socket was used for control channel send and receive operations. For testing purposes, the multicast group used for the control channel was at a fixed multicast IP address.

Four threads of execution were used to execute the tasks of the HyperCast protocol, in order to separate the different duties of the protocol and more efficiently handle packets without busy-waiting between packet arrivals. The different threads are as follows:

1. A “unicast monitor” thread that continually reads packets from the unicast socket, parses the packets, and places the messages into the incoming message queue.
2. A “multicast monitor” thread that continually reads packets from the multicast socket, parses the packets, and places the messages into the incoming message queue.
3. A “receiver” thread that continually reads messages from the queue and processes them as described in the event tables in Section 4.8.
4. A “pinger” thread which performs all the periodic actions of the protocol. The pinger thread cycles between two states: (1) sending *pings* to all neighbors in the neighborhood table, updating the time fields in the neighborhood table, and searching for tears in the neighborhood, and (2) sleeping for a period of time equal to $t_{\text{heartbeat}}$.

The use of the implementation for collecting performance statistics is discussed in Chapter 6.

5 HyperCast Verification

In this chapter the formal verification of the HyperCast protocol is presented. Neither simply specifying the protocol nor even testing an implementation of the protocol is guaranteed to reveal all defects in the protocol's design. Error conditions may be very difficult to detect and correct. A protocol verification tool was used to perform a formal verification of protocol correctness, thereby providing stronger evidence that the HyperCast protocol is free of logical defects.

5.1 The Need for Verification

As we recall from Chapter 4, in the HyperCast protocol each node has an associated state. The set of states and the rules governing the transitions between states defines a *finite state machine*. As each node moves from state to state, it keeps no memory of which states it has been in previously. The lack of memory is an advantageous property for a network protocol, since nodes do not have to store past information in order to operate. The task of implementing the protocol is greatly simplified since the only data that needs to be kept is the current state. However, being a finite state machine also means that there is no history of past messages processed at each node, so there is no means for a node to detect if it is caught in a repeating cycle.

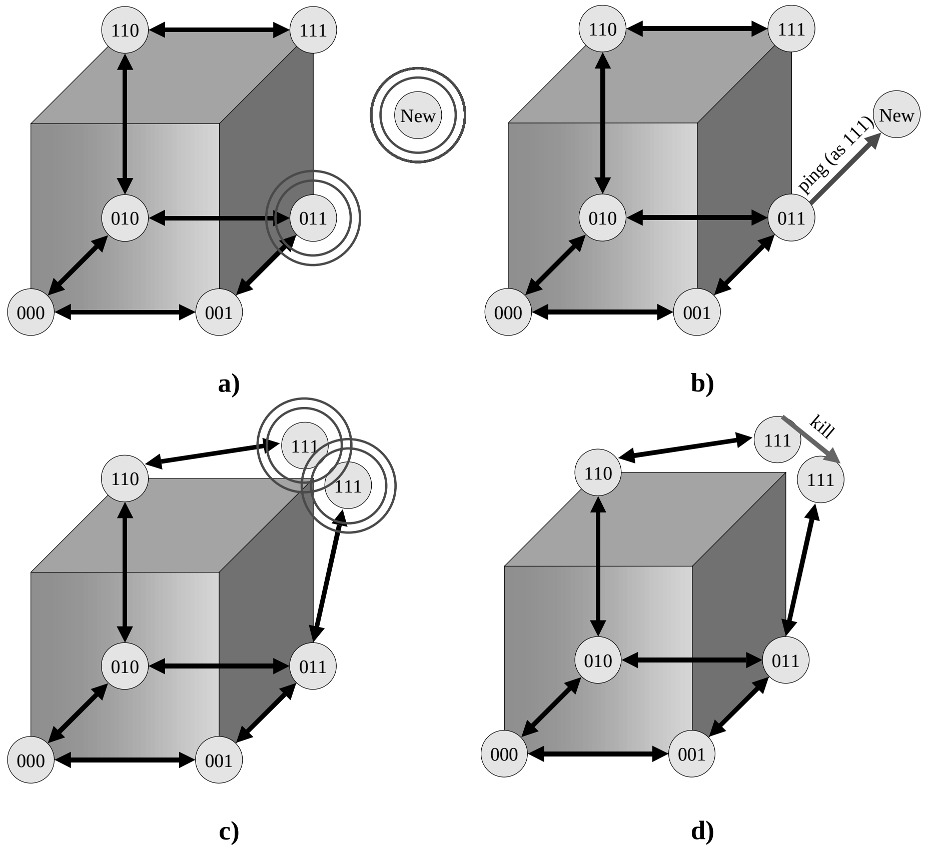


Figure 21: A cycle present in an earlier version of HyperCast.

For example, an earlier version of HyperCast did not make a distinction between the **Incomplete** and **Repair** states. The earlier version of HyperCast allowed nodes with incomplete neighborhoods to immediately send *pings* to the **HRoot** or **Joining** nodes to repair their neighborhoods, without first waiting for the timeout t_{missing}. A problem that can arise in that case is shown in Figure 21. Figure 21-a shows a possible state that the hypercube can reach, where the link between nodes "011" and "111" has caused dropped packets and node "011" has entered the **Incomplete** state. In the earlier version of HyperCast, nodes in the **Incomplete** state immediately sent *pings* to **Joining** or **HRoot** nodes to move them to a lower logical address in order to repair the tear in their neighborhood. In this case, the *beacon* from a **Joining** node is received first so the *ping* is sent to the **Joining** node to move the **Joining** node to the vacant logical address (Figure 21-b). The **Joining** node takes its position at “111”, however there is already a node that shares that same logical address. When the two “111” nodes receive each other’s *beacons* (Figure 21-c), the ***Duel*** mechanism eliminates one of them by sending a *kill* message to the node with the lower physical address (Figure 21-d). The eliminated node returns to the **Joining** state (Figure 21-a). Note that the hypercube has returned to its original situation, and the cycle can potentially continue indefinitely.

Protocol design flaws such as this may lead to *non-progress cycles*, where the protocol endlessly cycles nodes through states in a repeating manner while never reaching stability, or *deadlocks*, where nodes reach inconsistent states and cannot continue with protocol execution. *Race conditions* may also occur, where unpredictable behavior results from incorrect assumptions about process timing. Such problems are not trivial to detect, and may be hidden deep within the protocol description. Flaws can result in lengthy and complex error cases that cannot be found by human inspection. An additional problem with these types of errors is that they do not necessarily occur all the time, but only under certain (possibly rare) circumstances. A complex protocol such as HyperCast may suffer from any number of hidden protocol bugs which may escape normal testing.

Therefore it is not sufficient to simply claim that the HyperCast protocol works correctly based on the protocol description and protocol operation examples given in Chapter 4. A stronger assertion must be made, using a formal verification as evidence of protocol correctness.

5.2 Tool Description: Spin

A tool designed specifically for protocol verification developed at Bell Labs, called *Spin*, was employed to aid in the development of HyperCast [BEL98]. Spin checks the logical consistency of a protocol specification by searching for deadlocks, non-progress cycles, and any kind of violation of user-specified assertions [BEL98]. Spin has been used in many real-world applications, such as tracing design errors in various data communications protocols and concurrent algorithms.

To verify a design using Spin, the input model is specified using a programming language called the Process Meta Language (PROMELA). PROMELA is a non-deterministic programming language based on Dijkstra's guarded command language notation and Hoare's Communicating Sequential Processes (CSP) [DIJ76] [HOA78] [HOL97]. This input model is a complete representation of the specification of the protocol. In this case, the entire HyperCast protocol logic was encoded using PROMELA, as well as a system for simulating multiple hypercube nodes in a computer network.

Spin uses the model specified by the PROMELA code to construct a compact representation of the complete *state* of the system at any time. The system state uniquely identifies the condition of every part of the system at an instant in time. For the HyperCast protocol, the complete system state consists of the protocol states of each of the nodes in the simulated hypercube, combined with a list of all messages currently being passed between nodes. The effect of HyperCast protocol operations can then be described as transitions from one system state to another. The possible transitions between system states are governed by the actions of the HyperCast protocol.

Spin's ability to verify protocol correctness results from how Spin manipulates system state transitions. As a non-deterministic programming language, PROMELA has language constructs that allow for non-deterministic behavior. For example, in the PROMELA HyperCast implementation, incoming control messages passed are placed into queues at each node.

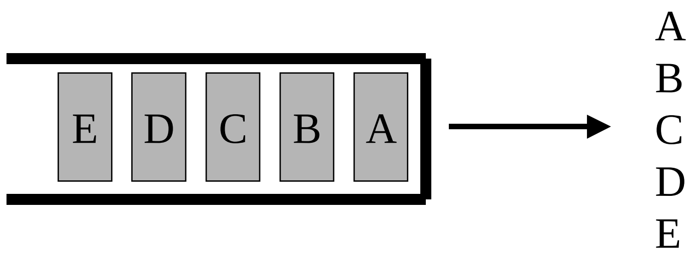


Figure 22: First-in, first-out queuing order in a deterministic language.

If the HyperCast implementation were written in a deterministic programming language such as C++ or Java, the code will most likely extract the messages from the queue and process them in first-in, first-out order (Figure 22).

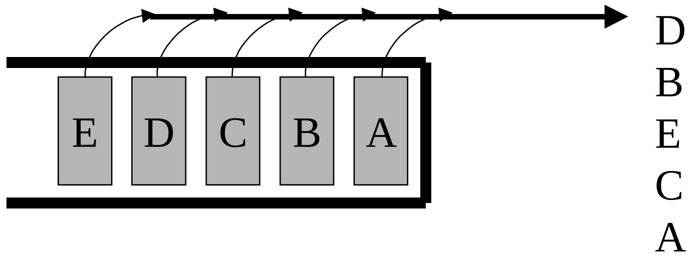


Figure 23: Random message selection in PROMELA.

In PROMELA, the same process of retrieving messages from the queue can be written instead as a basic language construct which extracts *randomly selected* messages from the queue (Figure 23). In a real network setting, the order in which a node receives messages is not guaranteed. Each possible random outcome of the random message retrieve operation represents a different valid possibility of how the protocol might execute in an actual application.

Thus at certain points in the code, Spin has a “choice” of which non-deterministic execution path to follow. It is through tracking these alternatives that Spin can verify protocol correctness. From any one system state, Spin can determine every possible state that the system can go into next. Thus by starting from an initial system state and sequentially selecting every possible next state, Spin can follow all execution paths and traverse the protocol’s entire state space. Traversing the entire state space means that Spin can visit and examine all possible states that the system can ever reach under the protocol’s direction.

Using this method, Spin can determine if a protocol has design errors. Potential protocol fault modes can be written in the form of assertions, which Spin checks while traversing the state space. Every possible route of execution will be examined by Spin, so the absolute worst cases (no matter how unlikely) can be considered and checked for assertion violations. A system state that has no further execution paths is deadlocked. Non-progress cycles are found when Spin traverses the state space and returns to a system state that is identical to one that has already been visited, thereby showing that the process of looping through states can continue indefinitely. When Spin finds these design flaws, it reports what type of error occurred and also which execution path led to the error. If every possible execution path leads to a correct result, then the protocol has been exhaustively verified to work correctly.

5.3 Verification of the HyperCast Protocol

The primary goal of the HyperCast verification with Spin was to show that in all cases the protocol will return an unstable hypercube to a stable state. To ensure that all cases were covered, non-deterministic clauses were used to represent any variability in the network. For example, in the case of the HyperCast message queues, non-deterministically selecting the next message to process ensures that the protocol works correctly without any assumptions on message ordering.

The initial state of the hypercube was represented using three parameters:

- The number of nodes joining a stable hypercube, J
- The number of nodes already present in the stable hypercube, N
- The number of nodes in the stable hypercube which fail unexpectedly, F

These parameters were varied to create a representative set of cases that the protocol has to deal with, over a range that will be described in Section 5.5. The state space traversal that Spin performs is memory intensive, since the number of possible system states for a hypercube is very large. These memory requirements increase exponentially with the number of nodes in the hypercube. Due to memory limitations, verification runs were performed using a maximum of six simulated nodes. Therefore the quantity $N + J – F$ was limited to being no more than six.

In addition to checking for deadlocks and non-progress cycles, Spin was used to ensure that every execution path resulted in a stable hypercube. This was accomplished by adding a method in the PROMELA implementation that detected when the system had reached a state corresponding to a stable hypercube. A stable hypercube was defined as when the nodes in the hypercube of size $N + J – F$ had unique logical addresses equal to

G(0) through *G*(*N* + *J* – *F* – 1), and all nodes were in the **Stable** state with the exception of the one **HRoot** at the highest logical address in the hypercube. Once stability had been reached, then that particular execution path safely terminated.

Alternatively, if the protocol had an error that resulted in the hypercube not always reaching a stable state, then a faulty execution path might run indefinitely. To check for this scenario, a method was added to the PROMELA implementation that detected if the system had been running for longer than a period of simulation time *T*, where *T* was an arbitrary large constant. This was represented as an assertion that the system time was always less than *T*. Spin found assertion violations and reported them if the assertion was violated along any execution path. If Spin's verification run results in no assertion violations and all execution paths result in successful terminations corresponding to stable hypercubes, then the protocol is proven to be logically consistent and the protocol meets the goal of always forming a stable hypercube.

Note that the assertion described above ensures that the hypercube will always reach a stable state within time *T*. Thus, *T* can be used as an upper bound on how long the protocol will take to reach stability. A useful performance measure is the exact amount of time that the protocol takes to achieve stability in the worst case. This exact value can be found by systematically reducing the constant *T* over successive verification runs, until *T* can be no lower without resulting in an assertion violation. This minimum *T* is thus the minimum upper bound on the time needed to reach stability. Other worst-case system measures were also found by using this method.

5.4 HyperCast Implementation in PROMELA

The mechanisms of the HyperCast protocol as described in Chapter 4 were encoded in PROMELA. Multiple hypercube nodes were simulated using distinct processes. PROMELA has no built-in language constructs to implement the passage of time, therefore a process was written which simulated a clock with time increments of $t_{\text{heartbeat}}$. With this clock functionality, the act of pausing a process for the next $t_{\text{heartbeat}}$ increment is accomplished by blocking until the clock variable is updated. When all such periodic functions are blocked, the clock process then increments the time and signals all the blocked processes to reawaken. This stepwise procedure ensures that processes all progress through time in a consistent manner, where no hypercube node is frozen in time while other nodes are not.

Additionally, a system was designed to simulate the network connecting the nodes together. Each hypercube node has an associated queue of inbound messages and a queue of outbound messages. The protocol process for each node removes random messages from the inbound queue, processes the messages, and places messages into the outbound queue. The network simulation process periodically removes random messages from the nodes' outbound queues and distributes them to the appropriate destination queues. If the outbound message is a *beacon*, the message is copied and distributed to the inbound message queues of all of the hypercube nodes. Otherwise, the destination physical address is read from the outbound message and the message is placed in the inbound queue of only the destination node.

5.5 Data

The protocol was tested to find the minimum upper bounds on time, number of unicast packets transmitted, and number of multicast packets transmitted given different initial system states. The total number of unicast packets transmitted and number of multicast packets transmitted were converted into average packet rates by dividing the total number of packets by the time. In the case of the unicast packet rate, the per-node average was found by dividing the aggregate unicast packet rate by the total number of nodes. Packet loss was not accounted for in this simulation, since probabilistic packet loss is difficult to model in a non-deterministic setting. Since Spin always finds the worst case scenario, with any chance of packet loss the worst case scenario is that all packets are lost (leading to a meaningless simulation).

Table 18: Verification Results: Time ($t_{\text{heartbeat}}$) vs. Number of JoiningNodes and Number of Nodes Present

Number of Nodes Present *N* \ Number of Joining Nodes *J*	0	1	2	3	4	5	6
0		7	11	16	19	20	23
1	1	5	7	10	11	14	
2	1	5	8	9	12		
3	1	7	7	10			
4	1	5	8				
5	1	7					
6	1						

Table 19: Verification Results: Unicast Packet Rate vs. Number of JoiningNodes and Number of Nodes Present

Number of Joining Nodes *J*

Number of Nodes Present *N*	0	1	2	3	4	5	6
0		0.0	0.4	0.6	0.8	0.8	0.9
1	0.0	0.4	0.6	0.8	0.8	1.0	
2	0.0	0.8	1.0	1.0	1.2		
3	0.0	1.3	1.2	1.3			
4	0.0	1.4	1.6				
5	0.0	1.7					
6	0.0						

Table 20: Verification Results: Multicast Packet Rate vs. Number of JoiningNodes and Number of Nodes Present

Number of Joining Nodes *J*

Number of Nodes Present *N*	0	1	2	3	4	5	6
0		0.7	1.1	1.6	2.1	2.8	3.2
1	0.0	0.8	1.6	2.1	2.8	3.2	
2	0.0	0.8	1.6	2.3	2.8		
3	0.0	1.0	1.7	2.2			
4	0.0	0.8	1.6				
5	0.0	1.0					
6	0.0						

Table 21: Verification Results: Time ($t_{heartbeat}$) vs. Number of Failed Nodes and Number of Nodes Present

Number of Failed Nodes *F*

Number of Nodes Present *N*	0	1	2	3	4	5	6
0							
1	1						
2	1	17					
3	1	23	17				
4	1	23	33	17			
5	1	38	44	36	17		
6	1	23	39	42	34	17	

Table 22: Verification Results: Unicast Packet Rate vs. Number of Failed Nodes and Number of Nodes Present

Number of Nodes Present *N* \ Number of Failed Nodes *F*	0	1	2	3	4	5	6
0							
1	1						
2	1	0.8					
3	1	1.4	1.6				
4	1	1.8	1.5	1.6			
5	1	2.0	1.7	1.8	2.5		
6	1	2.2	1.9	2.0	2.1	2.5	

Table 23: Verification Results: Multicast Packet Rate vs. Number of Failed Nodes and Number of Nodes Present

Number of Nodes Present *N* \ Number of Failed Nodes *F*	0	1	2	3	4	5	6
0							
1	1						
2	1	0.8					
3	1	1.5	0.8				
4	1	2.1	1.6	0.8			
5	1	2.0	1.7	1.2	0.8		
6	1	2.4	1.8	1.1	0.7	0.8	

5.6 Discussion

The data shows that the HyperCast protocol reliably returns an unstable hypercube to a stable state. Additionally, the performance measures found by the verification indicate that there are no worst cases that are significantly different from what can be expected given the protocol specification.

While verification cannot be used to prove results for large hypercube sizes, we assert that there is little qualitative difference between a hypercube of six nodes and a hypercube of six thousand nodes. It is unlikely that non-progress cycles and deadlocks exist in large hypercubes which do not have analogous fault modes in smaller hypercubes.

Not only did the verification tests provide evidence that the HyperCast protocol was working correctly, they also guided debugging efforts. For example, the cycle presented at the beginning of this chapter (Figure 21) was discovered and corrected using Spin. If it were not for the exhaustive search capability of the verification tool, it is possible that this cycle might not have been found.

6 HyperCast Experimental Validation

To determine the scalability properties of the HyperCast protocol, a full implementation of the protocol was tested. The goal of this testing was to determine if there are quantitative aspects of the protocol which adversely affect how it can scale up to an extremely large number of group members. While the computer facilities available for the testing of the HyperCast protocol during this research were not adequate for the testing of millions of nodes in real-time, they supported up to a thousand nodes. It was assumed that if the data collected for up to a thousand nodes showed that HyperCast was scalable, then it was reasonable to conclude that the trend will hold for still larger group memberships.

The protocol was implemented in the Java programming language for maximum portability across platforms, as described in Chapter 4. The implementation was an exact port of the code written in PROMELA, which was verified to be free of logical inconsistencies.

The protocol testbed was the Centurion computer cluster at the University of Virginia, a cluster used primarily as a platform for distributed computing research and for computational tasks such as large-scale simulations. The part of the cluster used for this experiment consists of 64 computers, each one a 533 MHz DEC Alpha with 256 MB of RAM running Linux 2.0.35 [LEG99]. The Centurion cluster machines are connected with a 100Mbit/s switched Ethernet network. Up to 32 logical hypercube nodes were run on each physical computer in order to maximize the number of nodes that the computer resources were able to test.

The measures of performance that were analyzed were:

- The number of packets (unicast and multicast) transmitted
- The number of bytes (unicast and multicast) transmitted
- The time needed to return the hypercube to a stable state

These measures of performance represent two attributes of the protocol. First, the control traffic transmitted must be shown to be scalable with the size of the multicast group. If the control traffic increases linearly with the size of the group, then the protocol will not be able to support large group sizes. Second, we wish to show that the time needed to return the hypercube to a stable state is not dependent on the size of the existing group. This time relates to how quickly the HyperCast protocol can incorporate dynamic changes in group membership into the embedded trees used for reliability.

These performance measures pertain only to the construction and maintenance of the hypercube itself. The scalability benefits of using embedded trees within the hypercube are shown in Chapter 3, and these embedded trees can be utilized whenever the hypercube is in a compact state. The advantage of using the hypercube as the control topology is known, however the measures of performance of the HyperCast protocol are used to determine whether the process of maintaining the hypercube structure is also scalable to large group sizes.

In order to examine the above measures of performance of the HyperCast protocol in different scenarios, it was necessary to define parameters that are used to describe the relevant characteristics of each scenario. These characteristics describe the initial state of the hypercube, as well as the dynamic group membership changes that test the HyperCast protocol's ability to mend the hypercube. Four attributes of the system were selected:

- The number of nodes joining a stable hypercube, J

- The number of nodes already present in the stable hypercube, *N*
- The number of nodes in the stable hypercube which fail unexpectedly, *F*
- The packet loss frequency on the network, *L*

The first three attributes are the same as the ones used in the verification tests described in Chapter 5. However, the tests were repeated with the Java implementation even though the theoretical worst-case results were already known. This repetition was performed because experimental data can be collected for a much larger group of nodes than the small number available in the verification, thereby allowing a better analysis of scalability trends. The experimental data is also representative of a typical real-world scenario, rather than showing results for only the worst case. Packet loss was artificially introduced at nodes by randomly dropping received packets before they were processed, so that the experiments more accurately modeled how HyperCast performs on a network subject to packet loss.

Three experiments were designed to evaluate the performance measures over a selected range of attribute values, described fully in Sections 6.1 through 6.3:

1. The effect of the number of joining nodes was analyzed with respect to the size of the hypercube. Thus *J* and *N* were varied, while *F* and *L* were held constant at 0.
2. The effect of the number of nodes that fail unexpectedly was analyzed with respect to the size of the hypercube. Thus *F* and *N* were varied, while *J* and *L* were held constant at 0.
3. The effect of the rate of packet loss was analyzed with respect to the size of the hypercube. Thus *L* and *N* were varied, while *F* and *J* were held constant at 0.

A testing harness was written to automate individual HyperCast trials. This testing harness consists of two components. The first component is a “server” program that is run on all the cluster computers involved in the experiment. The server runs a user-specified number of logical hypercube nodes, which in the majority of the trials was set to 32. These logical hypercube nodes are fully distinct from one another and operate concurrently in separate threads, thereby providing the same functionality as if each node was running on a different physical computer. The server broadcasts its presence to the second component of the testing harness, the “control” program. The control program is a front-end interface executed on one computer that has the duty of managing all of the servers. The control program reads from a list of individual trials specified by (N, J, F, L) tuples, and creates initial state information for the subset of the logical hypercube nodes needed to execute the trial.

The control program then distributes the initial state information to the servers, where it is incorporated into the logical hypercube nodes. Using this method, a hypercube in any possible state can be immediately created for the purposes of testing. The control program sends a signal to the servers to notify them to begin protocol execution. Once the protocol begins execution, the control program monitors the state of the hypercube. The servers periodically report the current state of all the hypercube nodes to the control program, so that the control program can determine if the hypercube has reached stability. Once the individual trial’s termination condition has been reached, the control program sends a signal to the servers to stop protocol execution. The servers report network statistics back to the control program, where they are aggregated into the measures of performance listed above.

This process is repeated for each of the many individual trials in the three experiments.

6.1 Experiment 1: Effect of Number of Joining Nodes

6.1.1 Description

This experiment examines the effect of the number of simultaneously joining nodes upon HyperCast performance.

In this experiment, the number of nodes already present in the hypercube N was varied across multiple trials. N was set to values ranging from 0 to 512 in increments such that $\log_2(N)$ was close to uniformly distributed. This was performed by setting N equal to $2^{i/2}$ rounded to the nearest integer, where the index i ranged from 0 to 18.

The number of joining nodes J was also independently varied across multiple trials, with values ranging from 1 to 512. The distribution of the values of J was also chosen so that $\log_2(J)$ was close to uniformly distributed, in the same manner as the values of N were chosen.

At the start of the trial, the hypercube of N nodes was in a stable state and all J joining nodes entered simultaneously. The end of the experiment was defined as when the hypercube contained $N + J$ nodes and was in a stable state. The time until stability was reached was measured, as well as the unicast and multicast traffic averages over that period of time.

6.1.2 Data and Interpretation

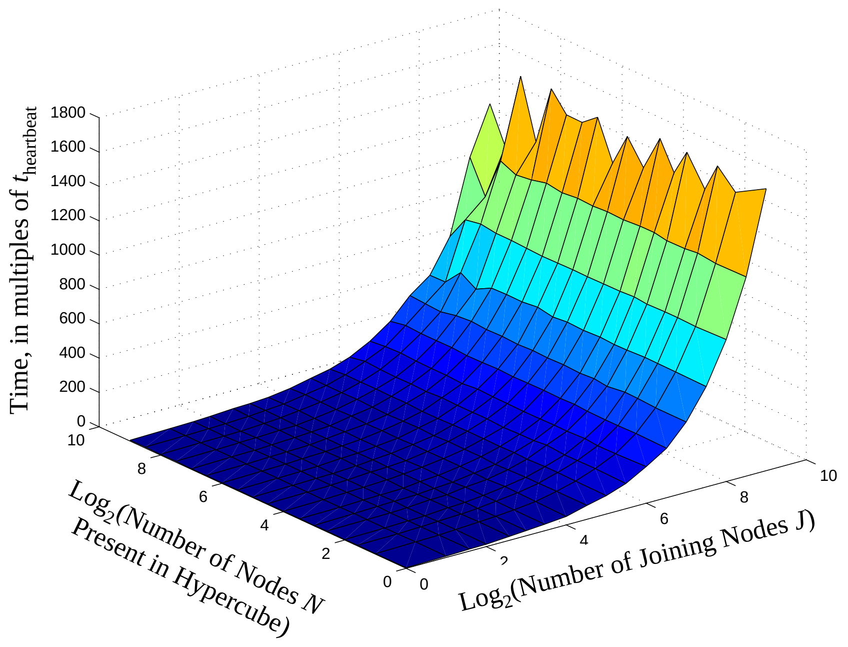


Figure 24: Time with respect to number of joining nodes and number of nodes present in hypercube.

Figure 24 shows the relationship between the amount of time needed to reach hypercube stability and the parameters *N* and *J*. Note that the plot indicates no correlation at all with the number of nodes present in the hypercube, which demonstrates scalability. The increase in time with respect to the number of joining nodes on the logarithmic axis indicates a linear correlation between the number of joining nodes and the time needed. This behavior is expected, since the process of adding one node to the hypercube takes a constant amount of time.

Note that the time is given in multiples of $t_{\text{heartbeat}}$. If a multicast group is relatively static with few members of the group joining or leaving during any given time period, then a relatively long value of $t_{\text{heartbeat}}$ can be chosen to minimize control traffic. If a multicast group expects more rapid changes in its membership, then a smaller value of $t_{\text{heartbeat}}$ can be chosen to trade protocol overhead for lower join latencies.

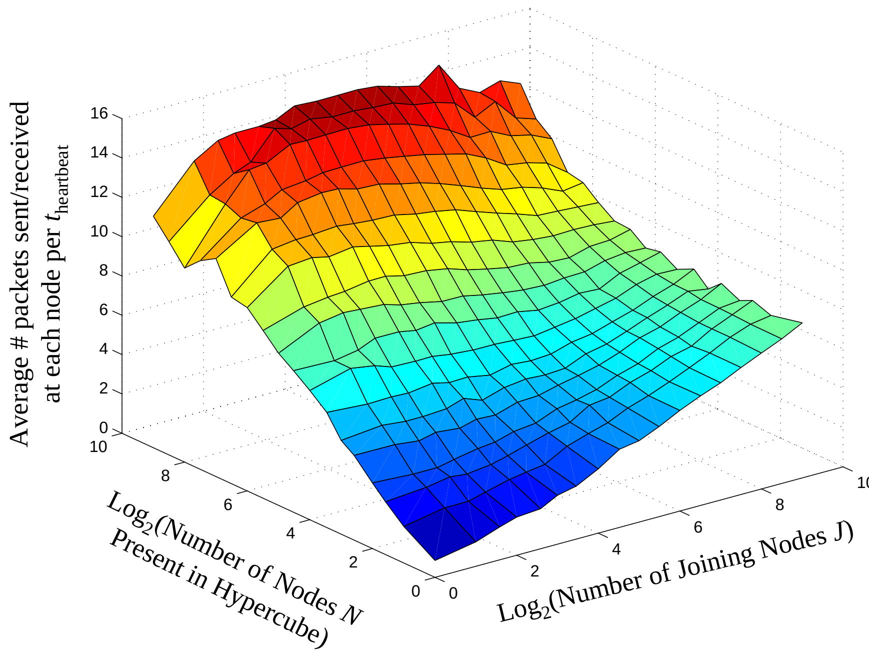


Figure 25: Per-node average unicast packet rate with respect to number of joining nodes and number of nodes present in hypercube.

Figure 25 shows the average unicast packet transmission rate during the time of the join operation. The data indicates that the unicast packet rate is approximately logarithmically related to both the number of nodes already present in the hypercube and the number of nodes joining the hypercube. The logarithmic correlation is present because the unicast transmissions are primarily *ping* messages sent between neighbors. The average number of neighbors of a node is approximately equal to the dimension of the hypercube, which is approximately equal to $\log_2(N + J)$. Hence the average number of *pings* sent should be proportional to $\log_2(N + J)$, a result supported by the data.

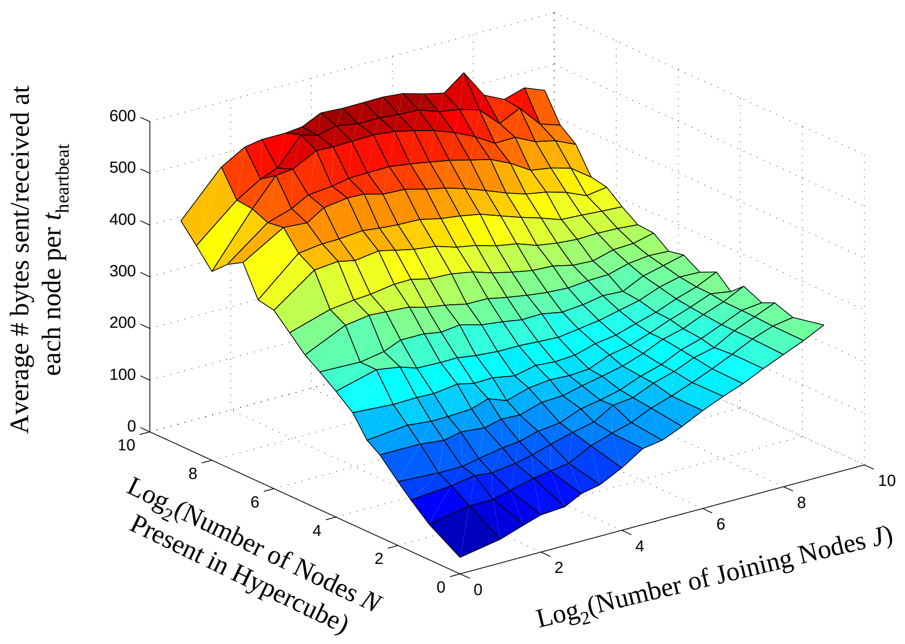


Figure 26: Per-node average unicast byte rate with respect to number of joining nodes and number of nodes present in hypercube.

Figure 26 shows the average unicast byte transmission rate during the time of the join operation. Since all HyperCast messages are of fixed length, this plot is equivalent to Figure 25. The byte rate figure is presented to show the approximate bandwidth used by the protocol, represented as bytes transmitted per $t_{\text{heartbeat}}$ interval.

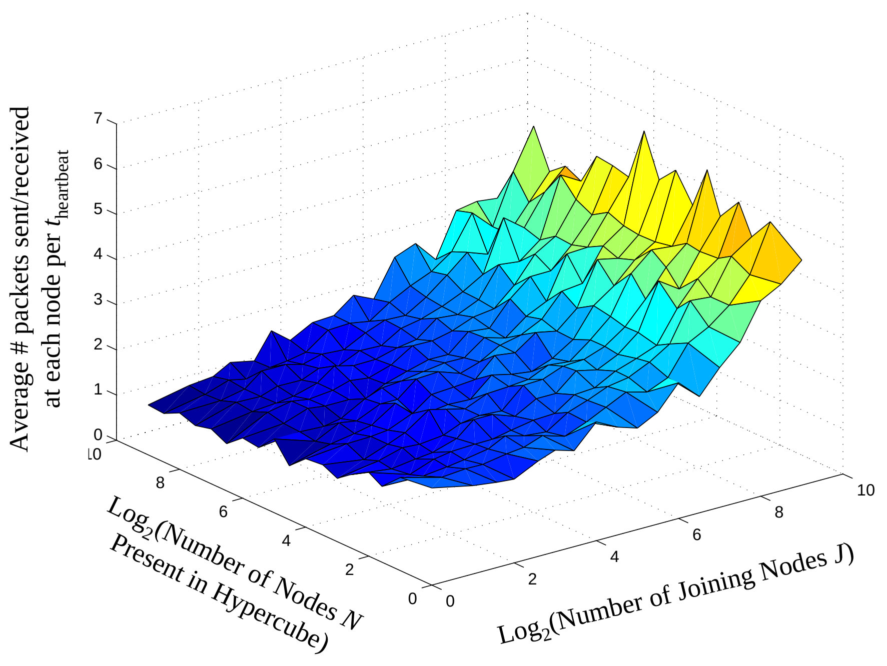


Figure 27: Multicast packet rate with respect to number of joining nodes and nodes present in hypercube.

Figure 27 shows the average rate of multicast transmissions sent and received at each node during the time of the join operation. The data indicates that there is no correlation between multicast traffic and the number of nodes present in the hypercube. There is however a sub-linear correlation between the multicast traffic and the number of nodes joining the hypercube. This correlation is due to the *beacons* sent by the **Joining** nodes. Without a *beacon* suppression mechanism for **Joining** nodes as described in Chapter 4, the correlation will be linear due to the fact that each **Joining** node sends one *beacon* per $t_{\text{heartbeat}}$ interval. The suppression mechanism described in Chapter 4 is able to eliminate the vast majority of redundant *beacon* messages, however there are still some additional redundant *beacon* messages which contribute to multicast traffic as the group size increases.

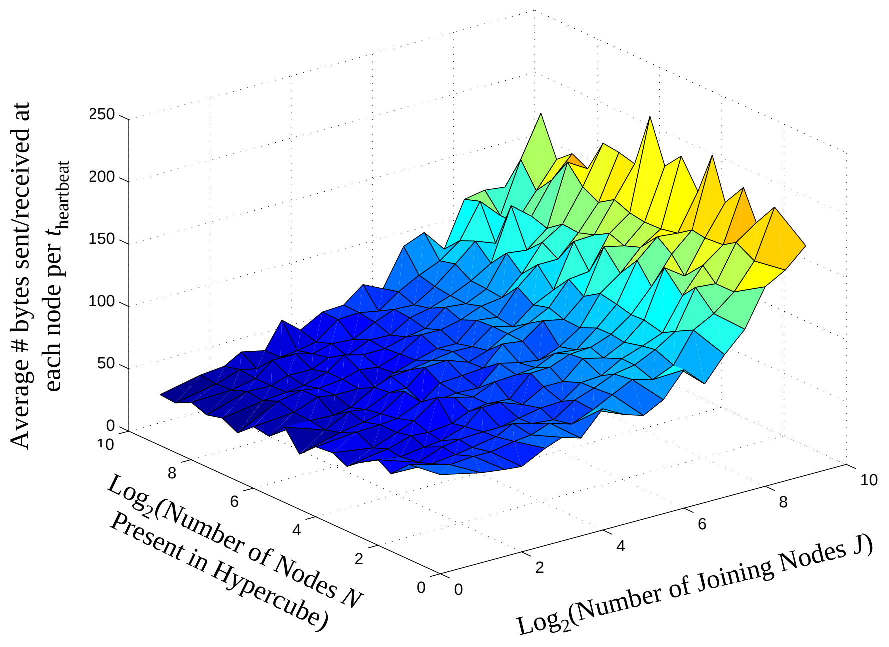


Figure 28: Multicast byte rate with respect to number of joining nodes and nodes present in hypercube.

Figure 28 shows the average multicast byte transmission rate during the time of the join operation. Since all HyperCast messages are of fixed length, this plot is equivalent to Figure 27. The byte rate figure is presented to show the approximate bandwidth used by the protocol, represented as bytes transmitted per $t_{\text{heartbeat}}$ interval.

6.1.3 Discussion

This experiment showed conclusively that the process of adding a node to the hypercube scales well to extremely large group sizes. Applications which require low latency in join operations can use a lower value of $t_{\text{heartbeat}}$, thereby reducing the time needed to add a node to the hypercube.

For future work, the HyperCast protocol can be improved somewhat by implementing more advanced suppression techniques used to suppress *beacons* from joining nodes, such as the proposed improvements to SRM discussed in Chapter 2.

Enhancing the suppression technique has the potential to reduce the amount of multicast traffic generated from large numbers of nodes joining at once.

6.2 Experiment 2: Effect of Number of Failed Nodes

6.2.1 Description

This experiment examines the effect of the number nodes that fail simultaneously upon HyperCast performance.

In this experiment, the number of nodes already present in the hypercube N was varied across multiple trials. N was set to values ranging from 0 to 512 in increments such that $\log_2(N)$ was close to uniformly distributed. This was performed by setting N equal to $2^{i/2}$ rounded to the nearest integer, where the index i ranged from 0 to 18.

The number of failed nodes F was also independently varied across multiple trials, with values ranging from 1 to N. The distribution of the values of F was also chosen so that $\log_2(F)$ was close to uniformly distributed, in the same manner as the values of N were chosen.

At the start of the trial, the hypercube of N nodes was in a stable state and all F nodes failed simultaneously. The end of the experiment was defined as when the hypercube contained $N – F$ nodes and was in a stable state. The time until stability was reached was measured, as well as the unicast and multicast traffic averages over that period of time.

6.2.2 Data and Interpretation

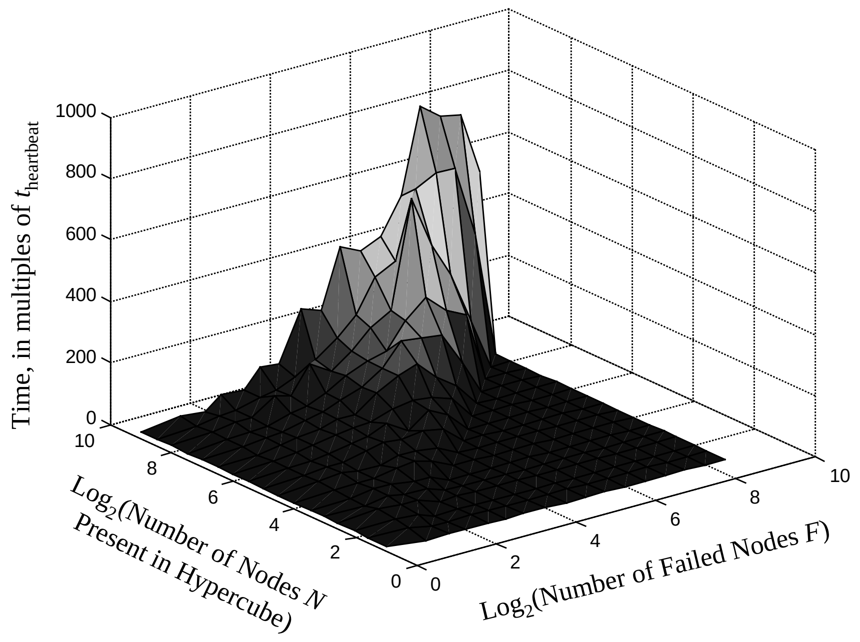


Figure 29: Time with respect to number of failed nodes and number of nodes present in hypercube.

Figure 29 shows the relationship between the amount of time needed to reach hypercube stability and the parameters *N* and *F*. Note that data was not collected for the degenerate cases where the number of failed nodes *F* is greater than the number of nodes present in the hypercube *N*. This fact is represented on the plot by the value 0 wherever $F > N$.

This plot shows no evidence of correlation between the time needed and the number of nodes present in the hypercube. Particularly, the plot is flat with respect to *N* at values of *F* where many data points exist to show trend information. The amount of time needed to restore hypercube stability appears to increase approximately linearly with the number of failed nodes. This result is expected, since the task of moving one node in the hypercube to a lower address takes constant time.

Note however that the data also indicates a slight reduction in the time needed to repair the hypercube as *F* approaches values close to *N*. The nature of the repair mechanism suggests an explanation for this non-linearity. If the number of failed nodes is close to the number of nodes in the initial stable hypercube, then the hypercube is sparsely populated due to the large number of node failures. In a sparsely populated hypercube, the protocol will have to move fewer nodes in order to reach a stable state. Therefore, the time needed to restore hypercube stability is actually linear with respect to the number of nodes which must be moved to lower addresses. The data corroborates this observation.

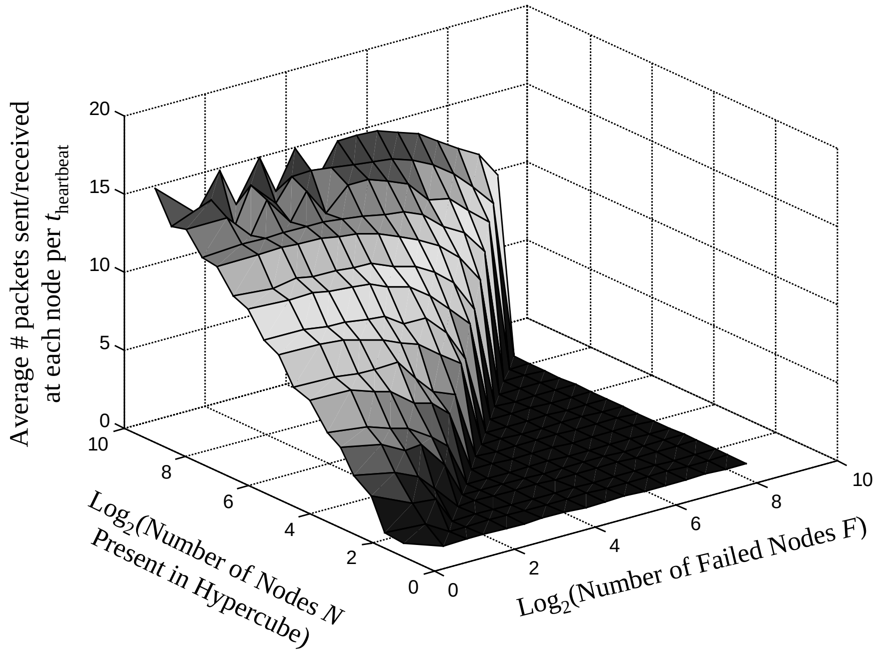


Figure 30: Per-node average unicast packet rate with respect to number of failed nodes and number of nodes present in hypercube.

Figure 30 shows the average unicast packet transmission rate during the time of the repair operation. The unicast transmissions are primarily *ping* transmissions, so the number of *pings* sent is proportional to the average number of neighbors of nodes in the hypercube. In this case, the hypercube contains $N - F$ nodes, therefore the average number of neighbors is approximately $\log_2(N - F)$. This relation is confirmed by the data.

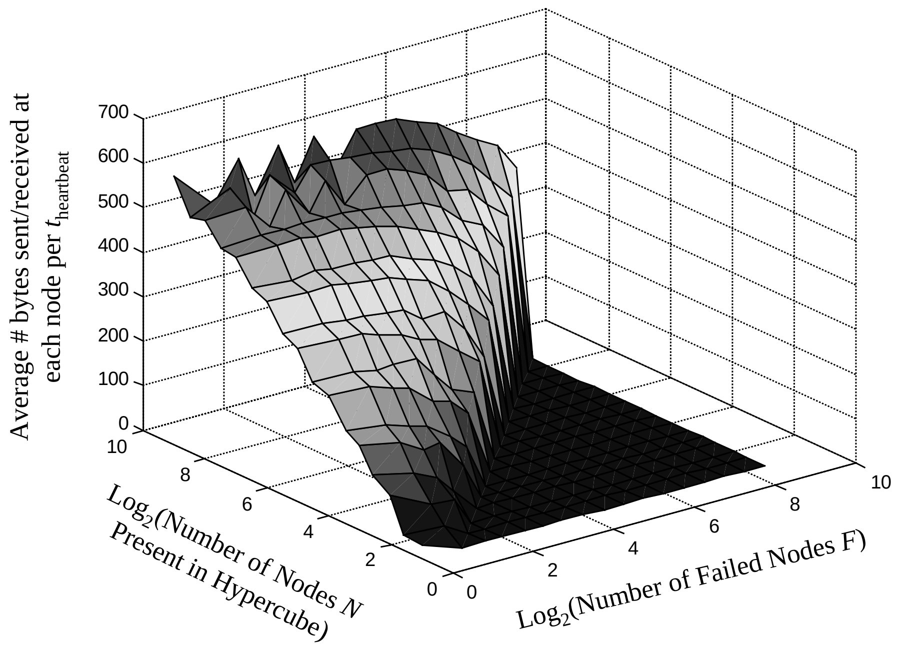


Figure 31: Per-node average unicast byte rate with respect to number of failed nodes and number of nodes present in hypercube.

Figure 31 shows the average unicast byte transmission rate during the time of the repair operation. Since all HyperCast messages are of fixed length, this plot is equivalent to Figure 30. The byte rate figure is presented to show the approximate bandwidth used by the protocol, represented as bytes transmitted per $t_{\text{heartbeat}}$ interval.

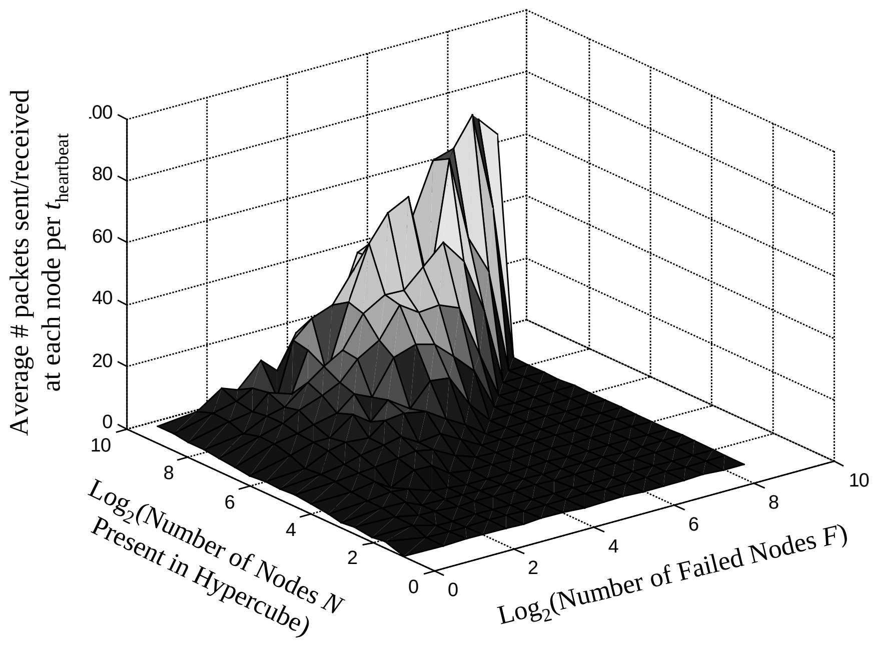


Figure 32: Multicast packet rate with respect to number of failed nodes and number of nodes present in hypercube.

Figure 32 shows the average rate of multicast transmissions sent and received at each node in the hypercube during the time of the repair operation. The rate of multicast transmissions is approximately linear with respect to the number of failed nodes, since the number of failed nodes is proportional to the number of tears in the hypercube that are created. For each tear in the hypercube, neighbors with incomplete neighborhood tables periodically send *beacon* messages, thereby contributing to the rate of multicast transmissions. The rate of multicast transmissions is also logarithmically related to the size of the hypercube. This relation is present because hypercubes of higher dimensions have more neighbors per node, and all the neighbors of a failed node send *beacons*.

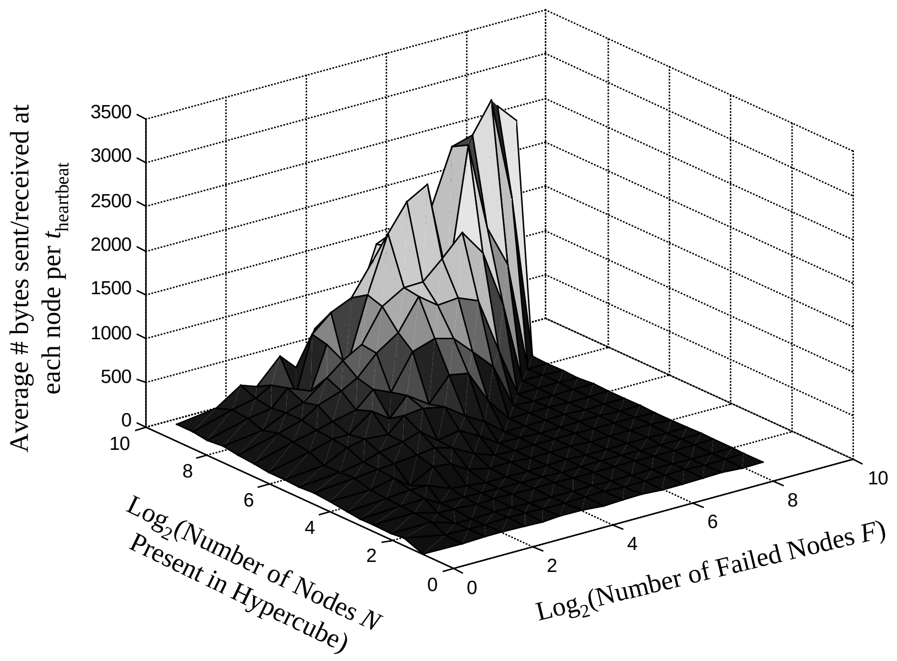


Figure 33: Multicast byte rate with respect to number of failed nodes and number of nodes present in hypercube.

Figure 33 shows the average multicast byte transmission rate during the time of the repair operation. Since all HyperCast messages are of fixed length, this plot is equivalent to Figure 32. The byte rate figure is presented to show the approximate bandwidth used by the protocol, represented as bytes transmitted per $t_{\text{heartbeat}}$ interval.

6.2.3 Discussion

This experiment indicates that as the size of a hypercube increases, the time required to repair a tear does not increase. This shows that the HyperCast protocol's repair operation scales well to large group sizes. Applications which require low latency in repair operations can use a lower value of $t_{\text{heartbeat}}$, thereby reducing the time needed to repair a tear in the hypercube.

Note that even though certain nodes may fail in a hypercube, there are many redundant links in the hypercube that may still be used for aggregation of control information. This feature of the hypercube may be utilized in a future implementation of the tree embedding algorithm which takes alternate routes into account, as will be discussed in Chapter 7.

6.3 Experiment 3: Effect of Rate of Packet Loss

6.3.1 Description

This experiment reveals the steady-state protocol overhead for different rates of packet loss.

In this experiment, the number of nodes already present in the hypercube N was varied across multiple trials. N was set to values ranging from 1 to 1024 in increments such that $\log_2(N)$ was close to evenly distributed. This was performed by setting N equal to $2^{i/2}$ rounded to the nearest integer, where the index i ranged from 0 to 20.

The proportion of packet loss L was also independently varied across multiple trials, with values ranging from 0.0 to 0.3 in increments of 0.03.

At the start of the trial, the hypercube of N nodes was in a stable state. During the trial, all nodes experienced the same level of random packet loss. The end of the experiment was defined as when 100 $t_{\text{heartbeat}}$ intervals had elapsed. The unicast and multicast traffic averages over that period of time were measured.

6.3.2 Data and Interpretation

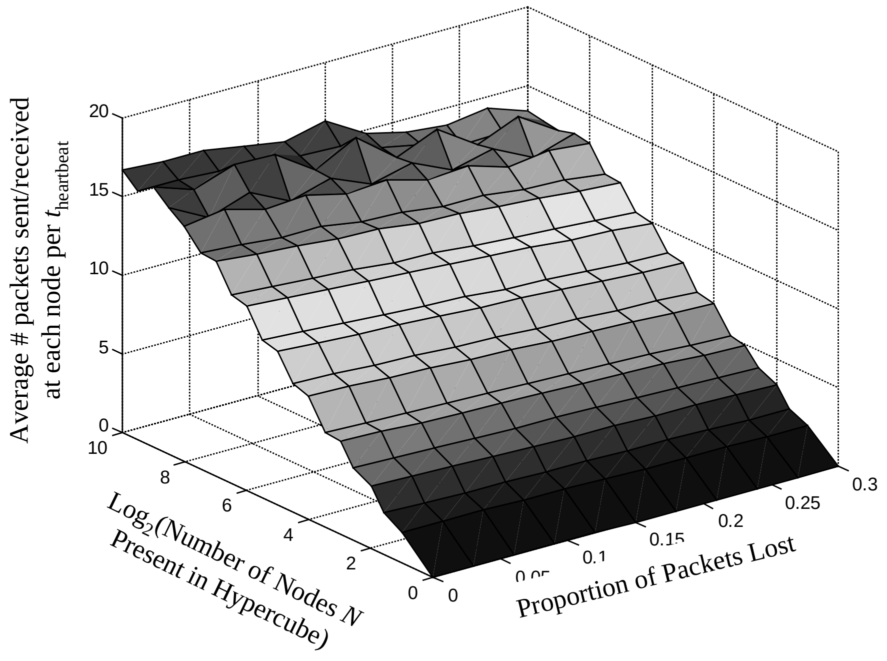


Figure 34: Per-node average unicast packet rate with respect to packet loss and number of nodes present in hypercube.

The plot in Figure 34 shows that as expected, the number of unicast packets sent and received is logarithmically related to the number of nodes in the hypercube. This behavior is due to the fact that the unicast messages are primarily *pings* sent between neighbors, and the average number of neighbors of a node is approximately $\log_2(N)$. Note also that there is a slight negative correlation between unicast transmissions and the proportion of packet loss. This negative correlation is due to a smaller proportion of packets being received as packet loss increases. The data indicates that as packet loss increases, the HyperCast protocol does not increase the frequency of unicast transmissions.

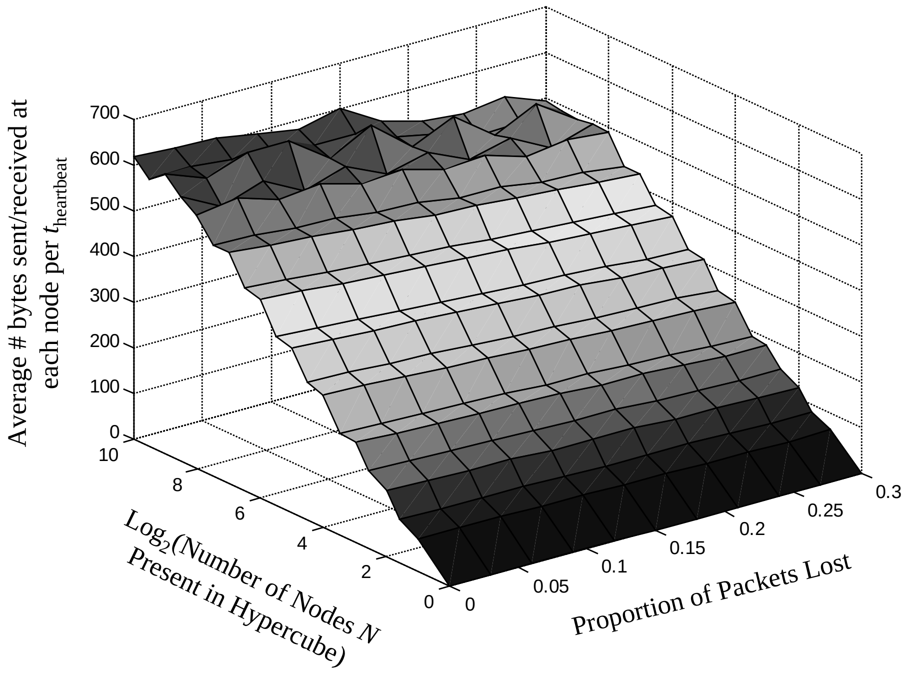


Figure 35: Per-node average unicast byte rate with respect to packet loss and number of nodes present in hypercube.

Figure 35 shows the average unicast byte transmission rate during the time of the trial. Since all HyperCast messages are of fixed length, this plot is equivalent to Figure 34. The byte rate figure is presented to show the approximate bandwidth used by the protocol, represented as bytes transmitted per $t_{\text{heartbeat}}$ interval.

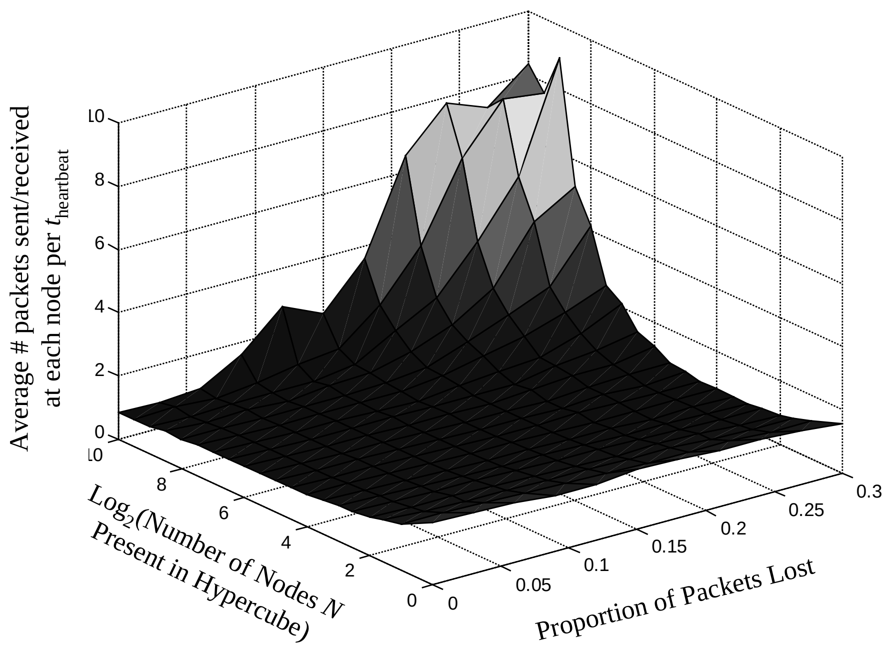


Figure 36: Multicast packet rate with respect to packet loss and number of nodes present in hypercube.

Figure 36 shows the average rate of multicast packets sent and received at each node during the time of the trial. At low packet loss rates, the protocol keeps a constant low multicast transmission rate. These multicast transmissions are the *beacons* sent periodically by the **HRoot**.

At higher packet loss rates, the number of multicast transmissions rises. Nodes often lose many consecutive *ping* packets when subjected to very high packet loss rates. These nodes then assume that one or more neighbors are missing, and so they broadcast *beacons* in attempts to contact their missing neighbors. The rise in multicast transmissions is also related to larger hypercubes, since larger hypercubes have more interconnections between nodes which are subject to packet loss. This behavior suggests that in applications which suffer from high packet loss rates, the variable t_{timeout} should be set to a higher value, thereby increasing the number of consecutive ping messages that must be dropped in order to create tears in the hypercube. By increasing the length of time t_{timeout} before entries in the neighborhood table become stale, the chance of packet loss causing a tear in the hypercube is reduced and therefore the number of multicast transmissions will also be reduced.

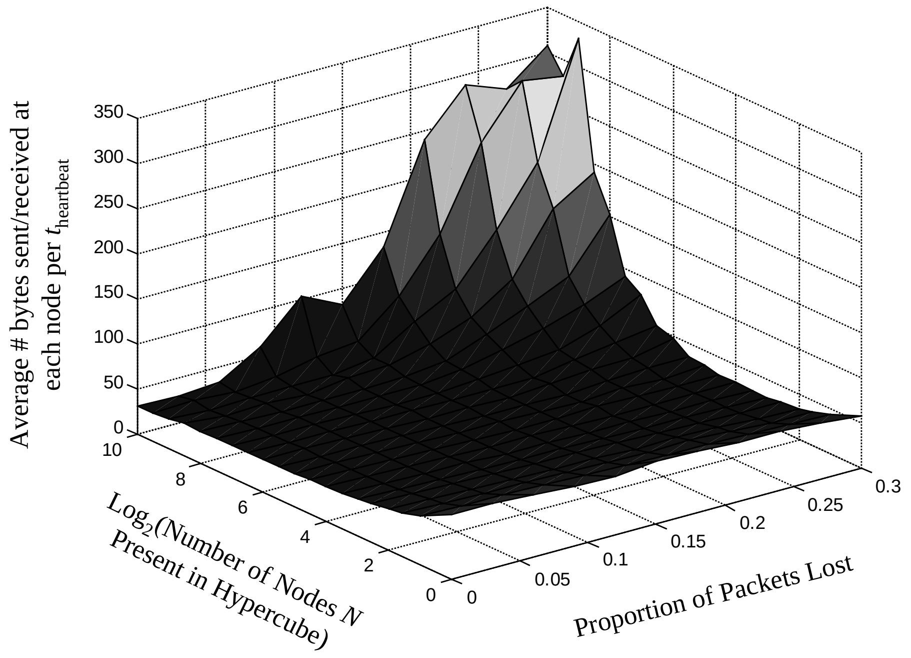


Figure 37: Multicast byte rate with respect to packet loss and number of nodes present in hypercube.

Figure 37 shows the average multicast byte transmission rate during the time of the trial. Since all HyperCast messages are of fixed length, this plot is equivalent to Figure 36. The byte rate figure is presented to show the approximate bandwidth used by the protocol, represented as bytes transmitted per $t_{\text{heartbeat}}$ interval.

6.3.3 Discussion

The primary concern about the HyperCast protocol revealed in this experiment is that multicast traffic grows quickly as packet loss increases. This increase in traffic growth can be reduced substantially by an adjustment in the protocol timeout values listed in Chapter 4. A simple analytical example shows that increasing the t_{timeout} value can have a strong impact on how many neighborhood table entries become stale due to dropped packets. Under a 20% rate of packet loss, if the number of consecutive *ping* packets from a neighbor that must be dropped to consider that neighbor's entry stale is raised by only five packets, then the chance of dropping enough consecutive packets to mark the neighbor's entry as stale is reduced by a factor of over 3000.

Note that at low packet loss rates (< 10%), the data indicates near-perfect scalability features of the protocol in steady-state operation. The multicast traffic is constant as group size increases, and the unicast traffic is only proportional to $\log_2(N)$. To put the logarithmic correlation in perspective, if the multicast group size is increased to a size of one million nodes instead of one thousand, the overhead of the average unicast traffic per node will only increase by a factor of two.

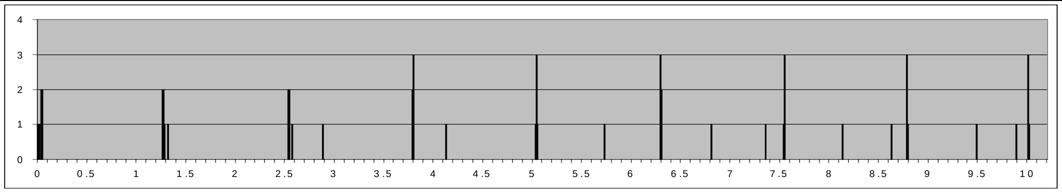
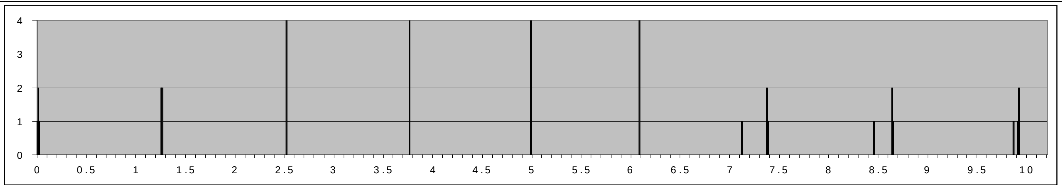
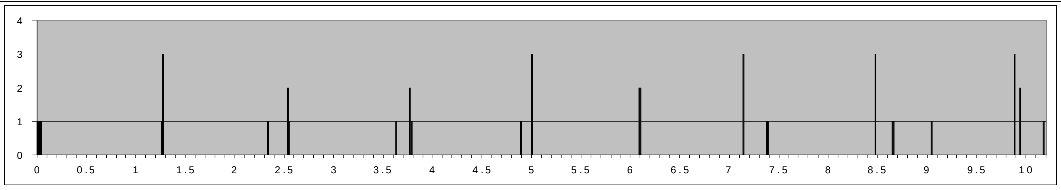
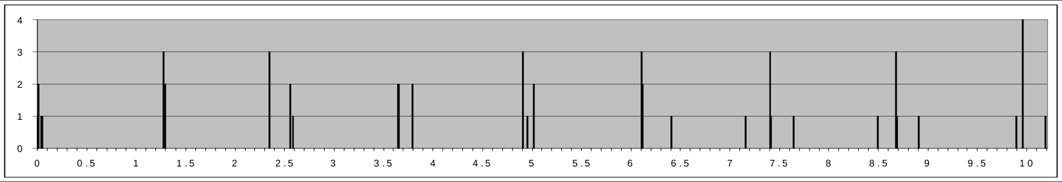
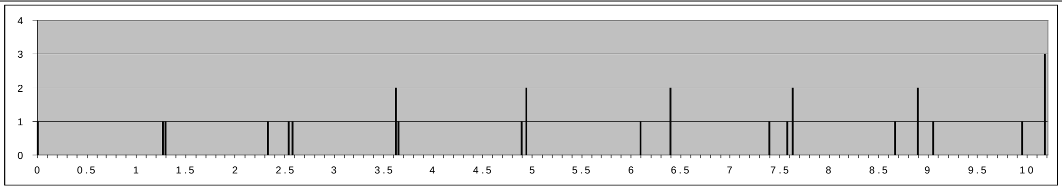
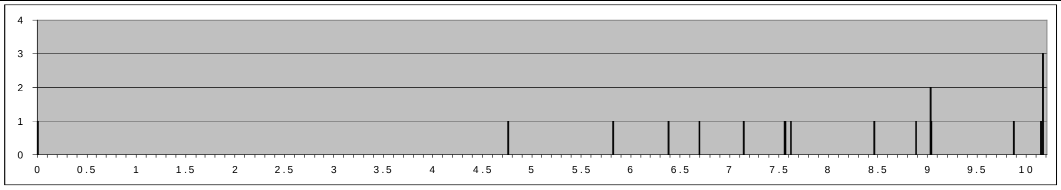
6.4 Examples of Transient Protocol Traffic

The performance characteristics examined in the experiments above presented aggregate network statistics. In this section, transient network statistics were collected in real time using the *tcpdump* utility on the Centurion cluster, which allowed for observations of the network traffic at the individual packet level.

6.4.1 Nodes Joining the Hypercube

A stable hypercube of four nodes was created, and two more nodes were added. The following table shows the transient packet transmissions for each of the nodes during the join operation, with time in multiples of $t_{\text{heartbeat}}$ in the x axis and the number of packets transmitted on the y axis.

Table 24: Transient data for join operation.

Logical Addr	Transient data
G(0)	
G(1)	
G(2)	
G(3)	
Join 1	
Join 2	

The large periodic spikes in the plots in Table 24 correspond to when the node is sending *ping* messages to all of its neighbors. *Ping* messages received from a node's neighbors also occur at regular intervals, however there is little correlation between the times at which different neighbors send pings.

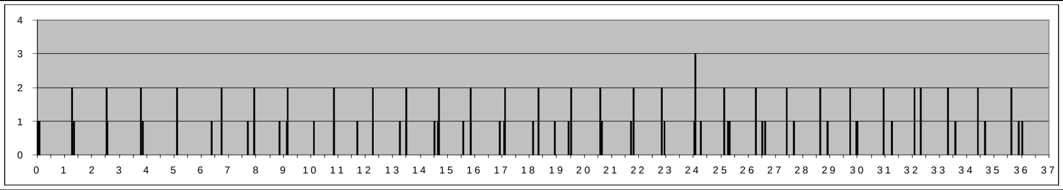
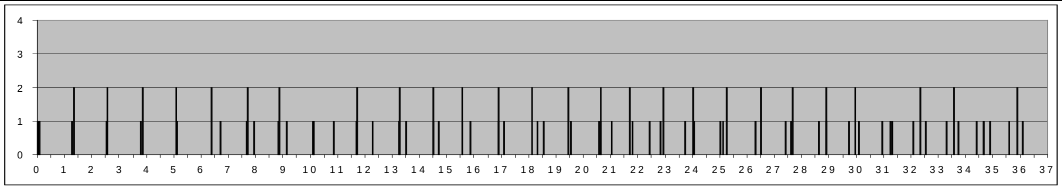
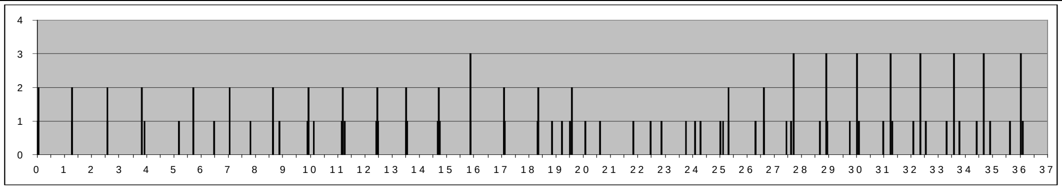
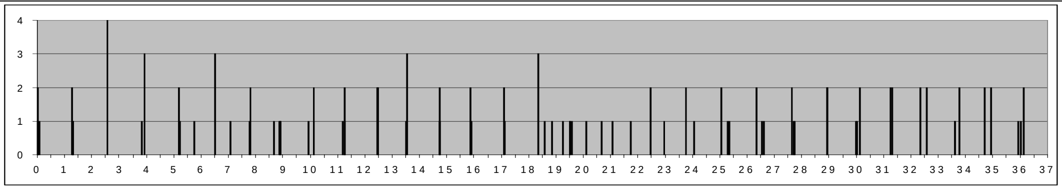
As the **Joining** nodes join the hypercube, they are entered into the neighborhood tables of the other nodes. This is shown in the table by the increasing length of the periodic spikes, indicating that the number of neighbors in the nodes’ neighborhood tables has increased.

The second **Joining** node shows a lack of traffic until approximately 5 $t_{\text{heartbeat}}$ units into the experiment. The lack of traffic is due to the suppression of the second node’s *beacons* by the reception of a *beacon* from the first **Joining** node.

6.4.2 Repairing Tears in the Hypercube

A stable hypercube of six nodes was created, and two nodes were set to fail. The following table shows the transient packet transmissions for each of the nodes during the repair operation, with time in multiples of $t_{\text{heartbeat}}$ in the x axis and the number of packets transmitted on the y axis.

Table 25: Transient data for repair operation.

Logical Addr	Transient data
$G(0)$	
$G(1)$	
$G(4)$	
$G(5)$	

The periodic spikes in Table 25 correspond to the *ping* messages sent by each node to its neighbors. While these plots do not clearly indicate the progression of the repair operation, some details are apparent. Note that the nodes at logical addresses *G*(4) and *G*(5) send a rapid succession of packets at approximately time *t* = 18. The traffic at time *t* = 18 corresponds to the process of moving the **HRoot** at *G*(5) to a lower logical address after the tears in the hypercube have been detected. Once the node at *G*(5) is moved, the node at *G*(4) is moved to fill the second tear. The relocated nodes rebuild their neighborhoods and again send *ping* messages to their neighbors, as indicated at times after *t* = 25.

7 Conclusions

In this research, the design, specification, verification and evaluation of the HyperCast protocol have been presented. The design and specification of the HyperCast protocol provide a simple, elegant set of messages and protocol states that operate efficiently and are relatively easy to implement. The formal verification has shown that the HyperCast protocol is free of logical inconsistencies, providing strong evidence to support the ability of the protocol to always return an unstable hypercube to a stable state. Evaluation of the protocol has shown that the HyperCast protocol is capable of efficiently building and maintaining the hypercube structure. The implementation has been tested for group sizes of up to 1024 nodes, and the data indicates that larger group sizes may be easily reached. Thus the HyperCast protocol is scalable to extremely large groups of users.

The logical hypercube topology created by the HyperCast protocol supports the efficient creation of embedded spanning trees, which can be rooted at any node. This spanning tree can be used for the aggregation of control information, preventing the implosion problem. The use of the hypercube and its embedded spanning trees for multicast groups with multiple senders has been shown to have theoretical performance advantages over the use of the current state of the art control topology, a shared K-ary tree. Thus the HyperCast protocol is capable of providing a control topology to multicast applications which processes control information more efficiently than existing solutions.

Therefore the impact of this research is that network applications which require the distribution of data between a large number of data sources and a large number of data recipients can make more efficient use of their network resources by using HyperCast.

Applications which use HyperCast may be able to support a much larger group membership than if HyperCast is not used. For example, a collaborative document editing tool with all group members sending data may only scale up to a few dozen simultaneous users using a tree-based reliable multicast protocol, however it may be able to scale up to a few hundred simultaneous users or more by using HyperCast. Therefore HyperCast adds value to computer applications by allowing them to provide a better service to the user.

7.1 Future Work

The HyperCast protocol currently organizes nodes into a hypercube structure and has been tested thoroughly with that goal in mind. However, the implementation does not make use of the embedded trees to aid in transmitting data at this time. This research will be continued to include a reliable multicast protocol with HyperCast, so that the embedded trees are used for the scalable processing and aggregation of control information. In the literature reviewed in Chapter 2, many such reliable multicast protocols were presented. It is likely that the future work of overlaying a reliable multicast protocol upon the embedded trees will be based on an existing protocol such as RMTP. The combination of HyperCast with a reliable multicast protocol can then be put to use with some basic applications, such as a simple multicast file transfer protocol. Performance measurements of these basic applications will be able to show the advantage of the hypercube control topology as compared to alternative control topologies, in a true real-world implementation.

Many of the control topologies presented in Chapter 2 are designed to attempt to correlate the logical topology with the network topology. For example, by creating a logical tree structure so that it corresponds with the multicast distribution tree, parent and child nodes are likely to be in close proximity to each other on the physical network. Therefore the local operations of passing control messages between logical parent and child nodes are also localized on the physical network, reducing the use of network resources. Additionally, packet losses are likely to be correlated within subtrees of the logical topology, so the aggregation of control messages within subtrees efficiently reduces redundant control information.

The HyperCast protocol currently makes no provision for placement of nodes in the hypercube according to physical layout. A possible path of future research on HyperCast is to develop a system by which nodes residing in close proximity to each other form independent hypercube control topologies. These independent hypercubes contain just the nodes in their respective physical domains. By selecting hypercube subgroups based on physical proximity, it is more likely that the logical links between nodes in the hypercubes correspond to shorter physical distances on the network. Elected representatives from each domain are in turn organized into a hypercube. A tiered approach has the potential to maintain the scalability benefits of the hypercube structure for multicast groups with multiple senders, while incorporating the performance benefit of short physical paths between logical hypercube neighbors.

One advantage of the hypercube structure that has not been addressed is that the hypercube contains a large number of redundant paths. If a network failure occurs in a tree structure, all nodes in the subtree below the network failure are cut off from the root of the tree. While the same event may also occur in an embedded tree within the hypercube, nodes within the hypercube have an advantage in that they each can have multiple neighbors. The failure of a node's parent in an embedded tree does not necessarily mean that there is no path of logical links from the node back to the sender, since the node's neighbors may have alternate connections to the sender. Future work can be done on creating a method for using these redundant links to provide a fault-tolerant control topology.

The timing parameters of the HyperCast protocol are currently set to fixed, known values. It may be advantageous for the protocol to be able to modify these parameters in order to compensate for variable network traffic conditions. A set of heuristics based on measurements of network latencies and packet loss may allow the HyperCast protocol to determine the timing parameters that provide the best tradeoff between the speed of hypercube operations and the protocol's network overhead.

8 Appendix A: PROMELA Verification Source Code

(in electronic form)

9 Appendix B: Java Implementation Source Code

(in electronic form)

10 References

[AMM92] M. Ammar and L. Wu. "*Improving the Performance of Point to Multi-Point ARQ Protocols through Destination Set Splitting*". IEEE Infocom '92, pp. 262-271, May 1992.

[BAS97] D. Bassett. "*Reliable Multicast Services For Tele-collaboration*". University of Virginia. Unpublished Master's Thesis, 1997.

[BEL98] Bell Labs Research (spin_list@research.bell-labs.com). "*On-The-Fly, LTL Model Checking with Spin*". Information available at http://netlib.bell-labs.com/netlib/spin/whatispin.html.

[CAM98] M. Campione, K. Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet (Java Series).* Addison-Wesley Publishing, March 1998.

[CAS94] S. Casner. "*Are you on the MBone?*". IEEE Multimedia. Vol. 1, No. 2. Summer 1994.

[CHI98] D. Chiu, S. Hurst, M. Kadansky, J. Wesley, "*TRAM: A Tree-based Reliable Multicast Protocol*". Sun Microsystems Laboratories, July 1998.

[COM91] D. Comer, "*Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*", Prentice Hall, Englewood Cliffs, New Jersey. 1991.

[DEE89] S. Deering, "*Host Extensions for IP Multicasting*". RFC 1112, Internet Engineering Task Force. August 1989.

[DEE91] S. Deering. "*Multicast Routing in a Datagram Internetwork*". Ph.D. thesis, Stanford University, Palo Alto California, December 1991.

[DIJ76] E. W. Dijkstra. *A Discipline of Programming*, Prentice-Hall, N. J., 1976.

[FLO95] S. Floyd, V. Jacobson, S. McCanne, C.G. Liu, and L. Zhang. *"A Reliable Framework for Light-Weight Sessions and Application Level Framing"*, ACM SIGCOMM '95, Boston. August 30-September 1, 1995.

[GRO97] M. Grossglauser. "*Optimal Deterministic Timeouts for Reliable Scalable Multicast*". IEEE Journal on Selected Areas in Communications, vol. 15, no. 3, pp. 422-433, April 1997.

[HAN98] C. Hänle and M. Hofmann. "*Performance Comparison of Reliable Multicast Protocols using the Network Simulator ns-2*". Proceedings of the Annual Conference on Local Computer Networks (LCN), October 1998.

[HOA78] C. A. R. Hoare. "*Communicating Sequential Processes*". Communications of the ACM, 21 (8):666-677, August 1978.

[HOL97] G. J. Holzmann. "*The Model Checker SPIN*". IEEE Transactions on Software Engineering Vol. 23, No. 5., May 1997.

[KAS96] S. Kasera, J. Kurose, and D. Towsley. "*Scalable Reliable Multicast Using Multiple Multicast Groups*". UMass CMPSCI Technical Report 96-73, October 1996.

[LAK90] S. Lakshmivarahan and S. K. Dhall. *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*. McGraw-Hill, New York, 1990.

[LEG99] The Legion Group, University of Virginia (legion@virginia.edu). "*Legion: A Worldwide Virtual Computer*". Information at http://legion.virginia.edu/.

[LEE99] K. W. Lee, S. Ha, and V. Bharghavan. "*IRMA: A Reliable Multicast Architecture for the Internet*". To appear in IEEE Infocom '99.

[LEI92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman Publishers, San Mateo, 1992.

[LEV96A] B. Levine. "*A Comparison of Known Classes of Reliable Multicast Protocols*". Masters Thesis, University of California Santa Cruz, June 1996.

[LEV96B] B. Levine, D. B. Lavo, and J. J. Garcia-Luna-Aceves. "*The Case for Reliable Concurrent Multicasting Using Shared ACK Trees*". Proceedings of 4th ACM International Multimedia Conference (ACM Multimedia 96), November 1996.

[LEV98] B. Levine, S. Paul, and J. J. Garcia-Luna-Aceves. "*Organizing Multicast Receivers Deterministically by Packet-Loss Correlation*". Proceedings of 6th ACM International Multimedia Conference (ACM Multimedia 98), September 1998.

[LI98] D. Li and D. R. Cheriton. "*OTERS (On-Tree Efficient Recovery using Subcasting): A Reliable Multicast Protocol*". Proceedings of 6th IEEE International Conference on Network Protocols (ICNP'98), October 1998.

[LIE98A] J. Liebeherr and B. S. Sethi. "*Towards Super-Scalable Multicast*". Technical Report, Polytechnic University, CATT 98-121. January 1998.

[LIE98B] J. Liebeherr and B. S. Sethi. "*A Scalable Control Topology for Multicast Communications*". Proceedings of IEEE Infocom 1998.

[LIN96] J. Lin. and S. Paul "*RMTP: A Reliable Multicast Transport Protocol*". Proceedings of IEEE Infocom '96, March 1996.

[LIU97] C.-G. Liu, D. Estrin, S. Shenker and L. Zhang. "*Local Error Recovery in SRM: Comparison of Two Approaches*", USC Technical Report 97-648, January 1997.

[LUC98] M. T. Lucas. "*Efficient Data Distribution in Large-Scale Multicast Networks*". Ph.D. Dissertation, University of Virginia, May 1998.

[MAC94] M. Macedonia and D. Brutzman. "*MBone Provides Audio and Video Across the Internet*". IEEE Computer, pp. 30-36. April 1994.

[NON96] J. Nonnenmacher and E. W. Biersack. "*Reliable Multicast: Where to use FEC*". Proceedings of IFIP 5th International Workshop on Protocols for High Speed Networks, October 1996.

[NON98] J. Nonnenmacher and E. W. Biersack. "*Optimal Multicast Feedback*". Proceedings of IEEE Infocom '98, March 1998.

[PIN94] S. Pingali, D. Towsley, and J. Kurose. "*A Comparison of Sender-Initiated and Receiver-Reliable Multicast Protocols*". Proc. 1994 ACM Sigmetrics and Performance '94, pp. 221-230, May 1994.

[QUI94] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, 2nd edition, 1994.

[RAM87] S. Ramakrishnan and B. Jain. "*A Negative Acknowledgement with Periodic Polling Protocol for Multicast over LAN*". Proceedings of IEEE Infocom '87. pp. 502-511. March 1987.

[STR92] W. Strayer, B. Dempsey, and A. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley Publishing, July 1992.

[TAL95] R. Talpade and M. H. Ammar. "*Single Connection Emulation (SCE): An Architecture for Providing a Reliable Multicast Transport Service*". Proceedings of the IEEE International Conference on Distributed Computing Systems, June 1995.

[XU97] X. R. Xu, A. C. Myers, H. Zhang, and R. Yavatkar. "*Resilient Multicast Support for Continuous-Media Applications*". Proceedings of NOSSDAV 1997.

[YAV95] R. Yavatkar, J. Friffioen, and M. Sudan. "*A Reliable Dissemination Protocol for Interactive Collaborative Applications*". ACM Multimedia 1995, pp. 333-343. November 1995.

[ZAB96] Zabele, S, DeCleene, B, and Koifman, A, "*Reliable Multicast for Internet Applications*". Proceedings of the Technical Conference on Telecommunications R&D in Massachusetts, March 1996.