HyperCast:
A Protocol for Maintaining Multicast Group Members in a Logical Hypercube Topology*

Jörg Liebeherr[#]

Computer Science Department
University of Virginia
Charlottesville, VA 22903

Tyler K. Beam ♦

Microsoft Corporation
Redmond, WA 98052

Abstract

To efficiently support large-scale multicast applications with many thousand simultaneous members, it is essential that protocol mechanisms be available which support efficient exchange of control information between the members of a multicast group. Recently, we proposed the use of a control topology, which organizes multicast group members in a logical *n-dimensional hypercube,* and transmits all control information along the edges of the hypercube. In this paper, we present the design, verification, and implementation of a protocol, called *HyperCast*, which maintains members of a large multicast group in a logical hypercube. We use measurement experiments of an implementation of the protocol on a networked computer cluster to quantitatively assess the performance of the protocol for multicast group sizes up to 1024 members.

---

* This work is supported in part by the National Science Foundation under grants ANI-9870336 and NCR-9624106 (CAREER).

Corresponding Author: J. Liebeherr, jorg@cs.virginia.edu, Tel: +1-804-982-2228, Fax: +1-804-982-2214.

♦ Tyler Beam's work described in this paper was performed while he was with the University of Virginia.

1 Introduction

A major impediment for scalability of multicast applications is the need of multicast group members to exchange control information with each other. Consider, for example, the implementation of a reliable multicast service. A unicast protocol with a single sender and a single receiver requires the receiver to send positive (ACK) or negative (NACK) acknowledgment packets to indicate reception or loss of data. If the same mechanism is applied to large groups, the sender would soon be flooded by the number of incoming ACK or NACK packets; this is referred to as the *ACK implosion problem* [7]. Even though many techniques and protocol mechanisms have been proposed to improve the scalability of multicast applications by solving the ACK implosion problem (e.g., [8][21][25]), the problem of protocol support for large-scale multicast applications, especially with a large number of senders, is not solved, and scalability to thousands of users is currently not feasible [26].

We are pursuing a novel approach to the problem of scalable multicasting in packet-switched networks. The key to our approach is to organize members of a multicast group in a logical *n-dimensional hypercube*. By exploiting the symmetry properties in a hypercube, operations, which require an exchange of feedback information between multicast group members, can be efficiently implemented.

Note: We do not consider that the hypercube is used for data transmissions, even though this is feasible. The main use of the hypercube is to channel the transmission of control information, such as acknowledgments, to avoid the ACK implosion problem.

In a previous paper [19], we have shown that by labeling multicast group members as the nodes of a hypercube, we can almost trivially build so-called *acknowledgment trees* [33], which can avoid ACK implosion even in very large multicast groups. We also performed an analysis of the load-balancing properties of tree embeddings in a hypercube, and demonstrated that the trees embedded in a hypercube have excellent load-balancing properties.

In this paper, we will present protocol mechanisms needed to maintain a hypercube in a connectionless wide-area network, such as the Internet. We discuss the design of the protocol, called **HyperCast**, which organizes the members of a multicast group in a logical hypercube. We will show an evaluation of the scalability properties of the *HyperCast* protocol through measurements of a prototype implementation. We will demonstrate that the *HyperCast* protocol is able to maintain a hypercube in a multicast group with dynamically changing group membership. The protocol achieves scalability through a distributed soft-state implementation. No entity in the network has knowledge of the entire group. The protocol is capable to repair a hypercube which has become inconsistent through failures of group members, network faults and packet losses.

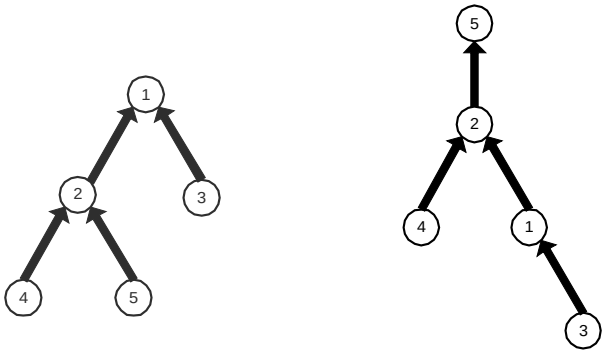
The approach presented in this paper is intended for *many-to-many multicast applications* where each group member can transmit information. There are many multicast applications where only one or few multicast group members actually transmit information, e.g., multicast web servers or electronic software upgrades. We do not claim that, in these situations, our approach presents any significant advantages over currently available solutions.

2 Control Topologies for Multicast Groups

In recent years, many protocol mechanisms have been proposed to solve the ACK implosion problem mostly in the context of providing a reliable multicast service (e.g., [1][2][5][6][7][9][10][11][13][14][16][17][18][22][23][24][29][30][31][32][33]). In packet-switching networks, we find two main approaches to limit the volume of control information which causes ACK implosion. In one approach, control information is broadcast to all or a subgroup of multicast group members, and a backoff algorithm [3][9] or a predefined bound on the volume of control traffic [28] is used is to avoid ACK

implosion. In the second approach, multicast group members are organized in a logical graph, henceforth called **control topology**, and every group member exchanges control information only with its neighbors in the logical graph. Control topologies that have been considered include rings [5][31] and trees [11][16][17][24][33].

Tree topologies have emerged as the most often proposed control topology. In a (proto-)typical tree topology, the members of a multicast group are organized as a *rooted spanning tree*, and all control information is transmitted along the edges of the tree. Tree topologies achieve scalability by exploiting the hierarchical structure of the tree. For example, by "merging" acknowledgments at the internal nodes of a tree, the number of acknowledgments received by a group member is limited to the number of children, thus, avoiding ACK implosion. In multicast applications with multiple senders, one tree is needed for each sender. Since maintaining a separate tree for each sender introduces substantial overhead, several protocols propose to use a single spanning tree, so-called *shared tree* [16][24] and "rehang" the tree with different nodes as root node.



(a) Node 1 is root (b) Node 5 is root

Figure 1. Re-hanging a shared tree with different nodes as root.

In Figure 1-a we show a binary tree control topology with node 1 as root. Figure 1-b depicts the same tree, "rehung" with node 5 as root node. Among the currently considered topologies, tree-based topologies seem to be most suited to support large multicast groups. In [19] we showed that, in multicast groups with a large number of senders, rehanging shared trees may not balance the load for processing control information across group members. We showed that a hypercube and tree-embedding algorithms in a hypercube (as shown in the next section) improve the load balancing properties.

3 Group Communication with Hypercubes

In this section, we describe the underpinnings of the proposed approach of using logical hypercubes to support group communications from [19]. 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 2.

We organize multicast group members as the nodes of a logical *n*-dimensional hypercube. By imposing a particular ordering on the nodes, we can efficiently embed spanning trees into the hypercube topology. By enforcing that control information between multicast group members can only be transmitted to the parent node in the spanning tree, the ACK implosion problem can be avoided. Since spanning trees serve the function of filtering acknowledgments transmitted to the root of the tree, the trees are referred to as *acknowledgment trees*.

Since, in an actual multicast group, the number of group members will not be a power of 2, we need to be able to work with hypercubes where certain positions are not occupied. We refer to a hypercube

with N nodes and $N < 2^n$ as an *incomplete hypercube*. For incomplete hypercubes we will try to maintain the following properties:

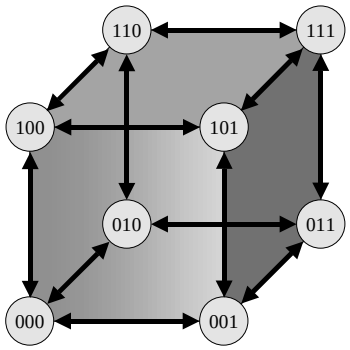


Figure 2. 3-dimensional hypercube with node labels.

Compactness: The dimension of the hypercube should be kept as small as possible, $n = \lceil log_2 N \rceil$.

Complete Containment of Trees: If we compute a spanning tree (acknowledgment tree) for an incomplete hypercube, we want to ensure that all nodes of the tree are present in the incomplete hypercube. (No node should be part of an acknowledgement tree if the node is not present in the cube.)

In a dynamic hypercube, compactness can be achieved by properly relabeling nodes whenever a node leaves the hypercube. Maintaining complete containment, however, is difficult to achieve, if the acknowledgment trees are computed in a distributed fashion and without global state information.

In [19], we presented a simple algorithm which guarantees complete containment of embedded trees. A key idea that leads to the algorithm is to use a *Gray code* [27] for ordering node labels of a hypercube and to add nodes to the hypercube in the order given by the Gray code. As an example, consider the labels of the 3-dimensional hypercube in Figure 2. If we add nodes to the hypercube, we need to have a rule for the order in which node labels are added. If we use the order of a binary encoding, we would add nodes in the order: $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow \ldots \rightarrow 111$. A Gray code would add node labels in the following order: $000 \rightarrow 001 \rightarrow 011 \rightarrow 010 \rightarrow \ldots \rightarrow 100$. In Table 1, we show the ordering of labels according to a binary code and a Gray code. It is worth noting that consecutive node labels using a Gray code differ in exactly one bit position.

Table 1: Comparison of Binary Code and Gray codes.

Index i =	0	1	2	3	4	5	6	7
Binary code: $Bin(i)$ =	000	001	010	011	100	101	110	111
Gray code: $G(i)$=	000	001	011	010	110	111	101	100

Using a Gray code, we can devise a simple algorithm, which embeds a spanning tree (acknowledgment tree) into an incomplete hypercube. The algorithm, given in Figure 3, is implemented in a distributed fashion: A node with label $G(i)$ calculates the label of its parent node in the tree with a root label $G(r)$, by only using the labels $G(i)$ and $G(r)$ as input. The algorithm consists of flipping a single bit.

The trees constructed by our algorithm have the following properties:

- **Property 1**. The path length between a node and a root is given by the Hamming distance of their labels.
- **Property 2.** If $N=2^n$ then the embedding results in a binomial tree.
- **Property 3.** In an incomplete and compact hypercube, the trees obtained by the algorithm are completely contained.

In Figure 8, we show two spanning trees generated by the algorithm for an incomplete hypercube with 7 nodes, for root nodes *000* and *111*.

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: $G(r) := R = R_n \ldots R_2 R_1$.

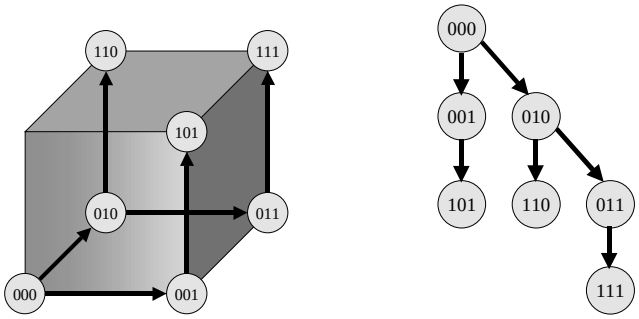
Output: *Label* of the parent node of node *I* in the embedded tree rooted at *R*.

```
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 ... I_k+1 (1 − I_k) I_k-1 ... I_2 I_1  with k = min_i(I_i ≠ 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 ... I_k+1 (1 − I_k) I_k-1 ... I_2 I_1 with k = max_i(I_i ≠ R_i)
    }
    EndIf
End
```

Figure 3: Tree Embedding Algorithm [19].

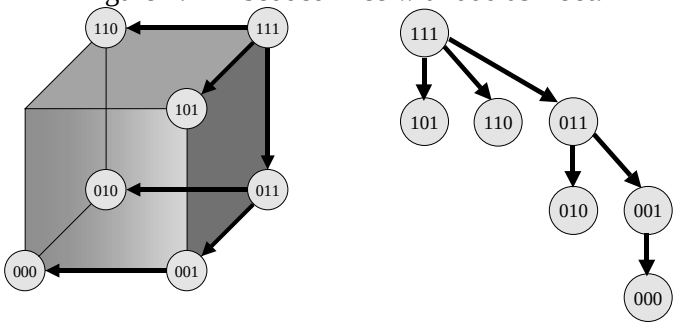
($G^{-1}(.)$ is the inverse function of $G()$ which assigns a number to a bit label, that is, $G^{-1}(G(k)) = k$.)

In [19] we performed an analytical comparison of the acknowledgment trees generated by the algorithm in Figure 3 and the acknowledgement trees generated in a shared tree approach (see Section 2). For both the hypercube and the shared tree, we assumed that spanning trees rooted at the sender are used for aggregation of control information. For the analysis, we made the simplifying assumptions, that (a) group communication is *symmetric*, that is, on the average each member of the group generates the same amount of control information, (b) the physical network topology is not considered, (c) the hypercube is complete, that is, $N = 2^n$, and (d) the number of nodes in the hypercube is constant. Under these assumptions, the hypercube was shown to have better load-balancing properties than a shared tree.[1] The analytical results have encouraged us to pursue the design and implementation of a protocol which maintains a hypercube control topology. We call this protocol **HyperCast**.



a) Embedded in hypercube b) Resulting tree

Figure 4: Embedded Tree with *000* as Root.



[1] Due to space considerations, the results from the analysis are not included in this manuscript.

a) Embedded in hypercube b) Resulting tree

Figure 5: Embedded Tree with *111* as Root.

4 The HyperCast Protocol

The goal of the *HyperCast* protocol is to maintain members of a multicast group as the nodes of a logical hypercube structure so that services, such as reliable multicast, can be implemented on top of the logical structure. It is emphasized that the *HyperCast* protocol is not concerned with transmission of data, nor does *HyperCast* provide any application-level services. The HyperCast protocol has mechanisms that allow new nodes to enter the hypercube, and it has procedures for repairing the hypercube in case of one or multiple failures.

The key to reach scalability of the logical hypercube to very large group sizes is that every node is aware of only a few nodes in the hypercube. No entity in the multicast group has complete state information.

4.1 Overview

The *HyperCast* protocol presented here takes advantage of the IP multicast service. A multicast group member, henceforth simply called a *node*, that wishes to participate in the hypercube structure joins a single IP Multicast group address, referred to as the *control channel*. Every node can both send and receive messages on this control channel. Obviously, scalability requirements demand that the traffic on this channel be kept minimal. We will see that only a few stations transmit to the control channel at a time.[2]

Nodes in the hypercube have a physical and a logical address. The *physical address* consists of the IP address of the host on which a node resides, and the UDP port used by the node for *HyperCast* unicast messages. Each node has a unique physical address. The *logical address* of a node is a bit string label, which uniquely indicates the position of the node in the hypercube (as discussed in Section 3). For an *m*-dimensional hypercube, *m* bits are needed for the logical address of a 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:

Consistent: No two nodes share the same logical address.

Compact: In a multicast group with N nodes, the nodes have bit string labels equal to $G(0)$ through $G(N-1)$.

Connected: Every node knows the physical address of each of its 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

The *neighbors* of a node in a hypercube are those nodes with logical addresses that differ from the logical address of the node in exactly one bit location. In an *m*-dimensional hypercube, every node has a maximum number of *m* neighbors. Every node maintains a table with the logical addresses of all its neighbors, the so-called *neighborhood table*. The fields for a single entry in the neighborhood table consist of:

- The neighbor's logical address,

[2] The protocol can be revised so that only a small subset of nodes is listening on the control channel at any given point of time. But, currently, we will assume that every member of the group is listening to the control channel.

- the neighbor's physical address, if it is known,[3] and
- the time elapsed since the node last received a message from the neighbor.

Given any node, its successor in the Gray's ordering is defined to be its *ancestor*. In a stable hypercube, every node except the one with the largest logical address has one ancestor. A node without an ancestor is defined to be a *Hypercube Root* (***HRoot***). In the HyperCast protocol, every node keeps track of the currently highest logical address in the hypercube (according to the Gray ordering), and assumes that this node is the *HRoot* (Note that this assumption may be incorrect in certain situations). The address of the highest known logical address is used by a node to determine which of its neighbors should be present in its neighborhood table. If, based on the highest address, a node determines that a neighbor should be present in the node's neighborhood table, but is not, the node is said to have an *incomplete neighborhood*. Each node keeps the following information on the node with the highest logical address: the logical address, the physical address, the time elapsed since it last received a message from this node, together with the last received sequence number from this node.[4]

In an instable hypercube, multiple nodes may consider themselves to be an *HRoot*. Also, different nodes in the hypercube may have different assumptions of who the *HRoot* is. In a stable hypercube, however, the hypercube has only one *HRoot*.

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 default values:

$t_{\text{heartbeat}}$ (default = 2s): Nodes send messages to each of their neighbors in the neighborhood table periodically every $t_{\text{heartbeat}}$ milliseconds.

t_{timeout} (default = 10s): When the time elapsed since a node last received a message from a neighbor exceeds t_{timeout} milliseconds, the neighbor's entry is said to be *stale* and the neighborhood table is said to be *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 t_{timeout}.

t_{missing} (default = 20s): As will be discussed in Section 4.4, after a neighbor's entry becomes stale, a node begins multicasting on the control channel to contact the missing neighbor. If the missing neighbor fails to respond for another t_{missing} milliseconds, the node removes the missing neighbor's entry from the neighborhood table and proceeds under the assumption that the neighbor has failed.

t_{joining} (default = 6s): Nodes that are in the process of joining the hypercube send multicast messages to announce their presence to the entire group. 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 backs-off from its attempt to join the hypercube for a period of time t_{joining}, before retrying to join.

4.4 Message Types

There are a total of four *message* types that are used by the *HyperCast* protocol. All messages are sent as short UDP datagrams. A node transmits a message, either by unicasting to one or all of its neighbors, or by multicasting on the control channel. We do not assume transmissions of basic messages to be reliable.

---

[3] If the physical address of neighbor is known at a node, we say that the neighbor is **present** in the neighborhood.

[4] The node with the highest logical address attaches sequence numbers to the multicast messages it sends, as will be discussed in Section 4.4. Nodes store this sequence number so that they can determine if they have received recent or outdated information.

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 of the currently known *HRoot*. A node transmits a *beacon* message only (1) if the node considers itself to be the *HRoot*, or (2) if the node determines that it has an incomplete neighborhood, or (3) if the node is in the process of joining the hypercube.

By construction of the hypercube, there is always at least one *HRoot*, and, therefore, at least one node is sending out *beacon*s on the multicast channel. In a stable hypercube, there is only one *HRoot*, and, thus, only one node sends out *beacon*s onto the multicast channel. Every node uses the *beacon* messages sent by *HRoot*(s) to form an estimate of the largest logical address in the hypercube. This information is sufficient to determine whether it has a complete neighborhood.

Each *beacon* message contains a sequence number, *SeqNo*, which is used to resolve conflicts if *beacons* are received from multiple nodes. The *HRoot*'s sequence number begins at 0. Whenever the *HRoot* sends a *beacon* message, *SeqNo* is incremented by one. Whenever a new *HRoot* is chosen, the sequence number is also incremented (*SeqNo* of new *HRoot* = *SeqNo* of current *HRoot* + 1). Since each node keeps track of the current *HRoot*, the sequence number tracks the timeliness of the information on the *HRoot*. When information at a node is not consistent, the information tagged with the lower sequence number is ignored.

The last group of nodes which send *beacon* messages are joining nodes which periodically send *beacons* to advertise their presence to the group.

Ping Message: Every node periodically sends a *ping* to all of its neighbors listed in its neighborhood table. A ping informs the receiver that the node is still present in the hypercube. A *ping* is a short unicast message, containing the logical and physical addresses 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 considers its neighborhood incomplete and begins sending *beacons* as described above. If it still has not received a *ping* from its neighbor after another period of time (t_{missing}), it assumes that its neighbor has failed and removes the missing neighbor from its neighborhood list. *Ping* messages are also used as the only mechanism to assign a new logical address to the receiver of a *ping* message.

Leave Message: When a node wishes to leave the hypercube, it sends a *leave* message to its neighbors. Nodes receiving this *leave* message remove the leaving node from their neighborhood tables. Since a *leave* message is not reliable, a neighbor may not receive a *leave* message. In this case, a neighbor will notice the absence of a neighbor through missing responses to its *ping* messages. Even without sending *leave* messages, a former neighbor eventually realizes that a node has left the neighborhood since no *ping* messages arrive for this neighbor.

Kill Message: A *kill* message is used to eliminate a node from the hypercube. More specifically, a *kill* message is used to eliminate nodes with duplicate logical addresses. A node which receives a *kill* message immediately sends a *leave* message to all its neighbors, and tries to rejoin the hypercube as a new node

4.5 Protocol Mechanisms

The *HyperCast* protocol implements two mechanisms for maintaining a *stable* hypercube. Recall from Subsection 4.1 that a stable hypercube satisfies the criteria for being *consistent*, *compact*, and *connected*.

Duplicate Elimination (Duel): The Duplicate Elimination ***(Duel)*** mechanism enforces consistency by ensuring that duplicate logical addresses are removed from the hypercube. 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. If the node's physical address is numerically greater than the conflicting node's physical address (according to the Gray 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.

Address Minimization *(Admin):* The Address Minimization ***(Admin)*** mechanism is used to maintain compactness of the hypercube. On a conceptual level, the *Admin* mechanism attempts to move nodes into lower logical addresses whenever opportunities arise. To see how *Admin* reconstitutes compactness, recall first that a hypercube which violates compactness must have a *tear* in the hypercube fabric (that is, some node has an incomplete neighborhood table). The *Admin* mechanism enforces that a node with a logical address higher than the logical address of a tear lowers its logical addresses to repair the tear. (Note: In a compact hypercube no node can move to a lower address).

The *Admin* mechanism at a node consists of an active and a passive part. The active part is executed, when a node receives a *beacon* message from the *HRoot* and the node realizes that it is missing a neighbor in its own neighborhood table which has a lower logical address than the *HRoot*. In such a situation, the node sends a *ping* with the missing lower logical address to the *HRoot*. The passive part is activated if the *HRoot* receives such a *ping* message with a destination logical address lower than its current logical address. In this case, the *HRoot* sets its logical address to the value given in the *ping*.

The *Admin* mechanism also governs the process of nodes joining the hypercube. Initially, the logical address of a newly joining node is marked as an invalid logical address. The invalid address is assumed to be larger than any valid address in the hypercube. Since a joining node 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 there is node with an incomplete neighborhood, this node sends a *ping* to the new node with the address of the vacant position. The new node assumes the (lower) address given in the *ping* message and occupies the vacant address. If there is no tear in the hypercube, the new node is placed as a neighbor of the *HRoot*. More precisely, the *HRoot* sends a *ping* to the new node containing the logical address which corresponds to the successor of the *HRoot* in the Gray ordering. Therefore, a node which joins a stable hypercube becomes the new *HRoot*.

The *Duel* and *Admin* mechanisms, respectively, enforce consistency and compactness of a hypercube. The last criterion for a stable hypercube, connectedness, 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 in the hypercube, the logical/physical address pair of node *B* is added into node *A*'s neighborhood table. If a neighbor does not send *pings* for an extended period of time, it is assumed that the neighbor has dropped out of the hypercube and its entry in the neighborhood table is removed. Actions taken by the *Admin* mechanism then repair the tear in the neighborhood table.

4.6 States and State Transitions

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.

In Figure 6, we show the state transition diagram of the *HyperCast* protocol. The states are indicated as circles. State transitions are indicated as arcs; each arc is labeled with a condition which triggers the transition. We refer to [20] for a more detailed description of the state transitions.

With the state definitions, we can give a precise definition of a stable hypercube. A hypercube with *N* nodes is stable if all of its nodes have unique logical addresses, ranging from $G(0)$ to $G(N\text{-}1)$ (where G(.) indicates the Gray code discussed in Section 3), and all nodes are in state *Stable*, with the exception of the node with a logical address $G(N\text{-}1)$ which is in state *HRoot*.

Table 2: 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 tjoining.
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 ttimeout, 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 ttimeout.
Repair:	Has been Incomplete for a period of time tmissing 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 hypercube.
HRoot/ Incomplete:	Incomplete node which believes that it has the highest logical address in the entire hypercube.
HRoot/Repair:	Repair node which also believes that it has the highest logical address in the hypercube.
Leaving:	Node that wishes to leave the hypercube.

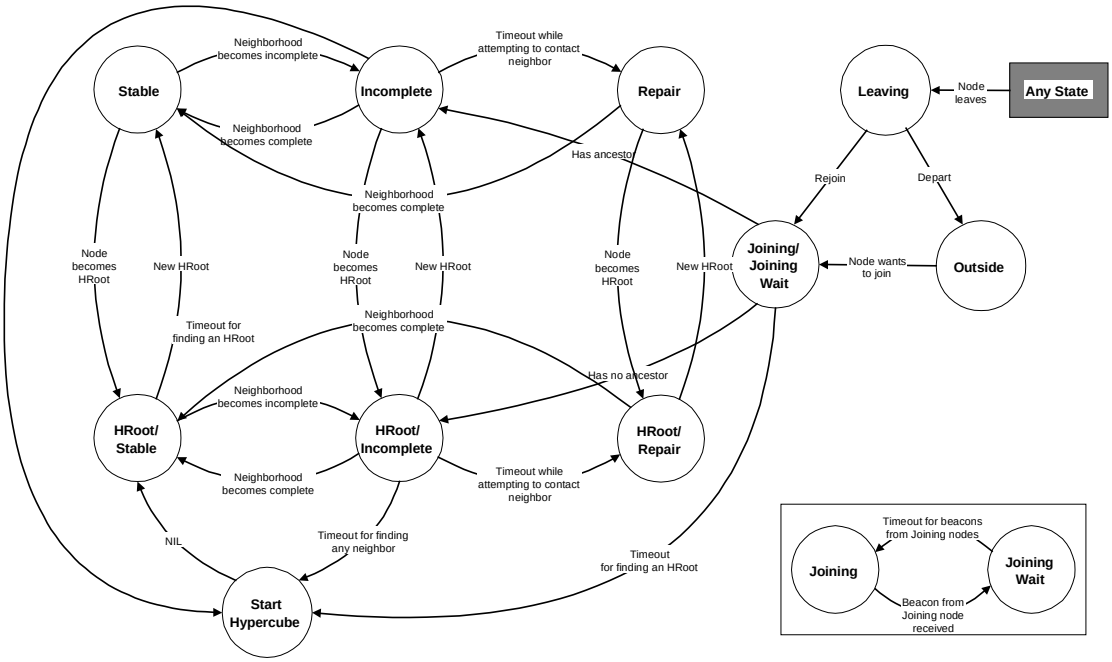


Figure 6: Node State Transition Diagram

4.7 Examples

We next illustrate the operations of the protocol in simple examples. In the examples, we use a small number of nodes and we assume that there are no packet losses.

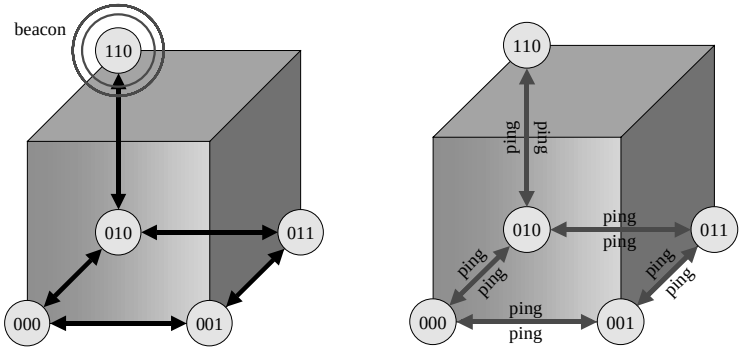


Figure 7: Stable hypercube.

4.7.1 The Stable Hypercube

Figure 7 shows a hypercube with five nodes, represented as circles. We use arrows to represent unicast messages. Circles around a node indicate a multicast message. In Figure 7-a, we show a stable hypercube. Here, the *HRoot*, node *110*, multicasts *beacons* periodically. The *beacon* is received by all nodes and keeps all nodes informed of the logical address of the *HRoot*. Therefore, the nodes know which of their neighbors should be present in their neighborhood tables. Every node periodically sends *ping* messages to its neighbors in the neighborhood table (Figure 7-b).

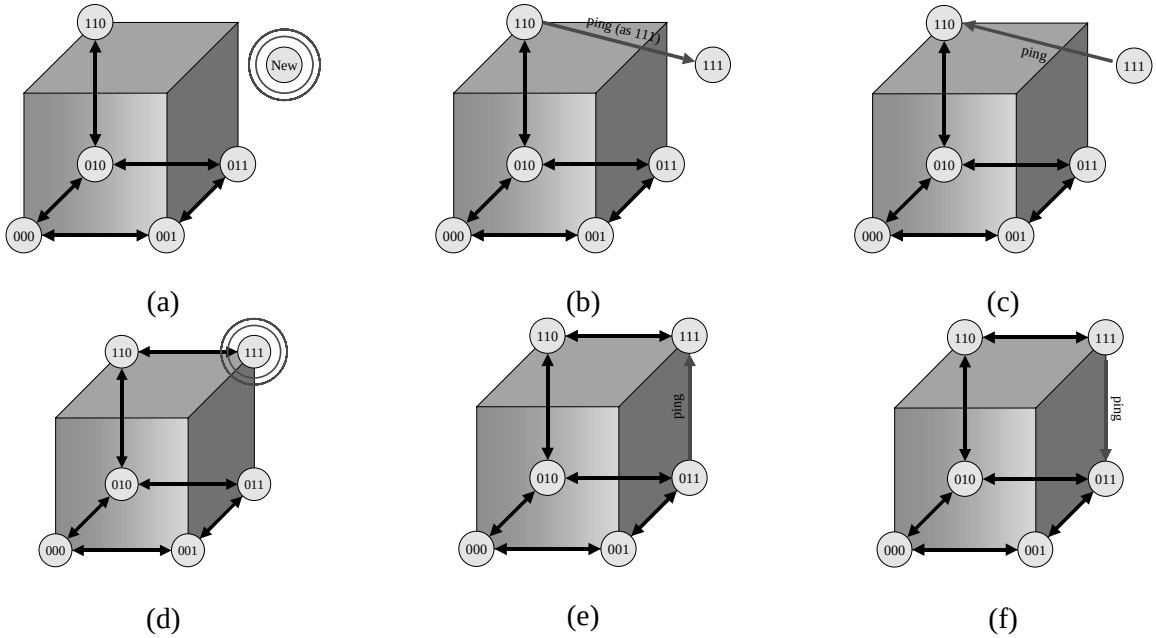


Figure 8: Joining node.

4.7.2 Adding a Node

In Figure 8-a**,** we show a node in state *Joining*, labeled "New", which wants to join the hypercube. The node periodically sends *beacon* messages, thus, making its presence known to the group. The *HRoot* places the *Joining* node as its neighbor at the next successive position in the hypercube according

to the Gray ordering, and *pings* the new node with the new logical address (*111*) (Figure 8-b). The new node takes on the new logical address and replies with a *ping* back to the original *HRoot* (Figure 8-c).

The new node determines from the *ping* packet that it is the *HRoot*, since its own logical address is equal to its highest known logical address. It begins sending *beacons* as an *HRoot* (Figure 8-d). If node *011* receives the *beacon* from the new *HRoot*, it realizes that *111* should be its neighbor. Thus, node *011* sends a *ping* message to *111* (Figure 8-e). Once node *111* receives the *ping* message, it responds with a *ping* itself (Figure 8-f). At this time, all nodes in the hypercube have complete neighborhood tables and know all their neighbors, so the hypercube is stable.

4.7.3 Repairing a Tear

We next describe the operations performed when a single nodes fails. Suppose, that node *001* in our example fails, i.e., it ceases to send messages or respond to messages (Figure 9-a). The neighbors of *001* detect the failure, since the neighbors do not receive *ping* messages from the failed node (Figure 9-b). After a time t_{timeout}, the neighbors of the failed node, *000* and *011*, start sending *beacons* to indicate that they have detected a missing neighbor (Figure 9-c). (If the failed node resumes operation, the *beacons* from its neighbors are used to reestablish the logical connections in its neighborhood table.) If, after sending *beacons* for a period of time t_{missing}, the failed node is still not responding, the neighbors assume that node *001* does not return and start trying to find a replacement. Then, the *Admin* mechanism is activated at nodes *000* and *011*, and the nodes try to make the current *HRoot* their neighbor. In Figure 9-d, we show that node *011* sends a *ping* to the current node *111*; the *ping* identifies node *111* as *001*. When *111* receives this *ping* messages, it realizes that it can reduce its logical address by assuming the label *001*. With the *Admin* mechanism, node *111* now leaves its current position in the cube, and fills the tear in position *001*.

After receiving the *ping* from node *011*, node *111* sends *leave* messages to its neighbors to notify the neighbors that it will leave its current neighborhood, and assumes the logical position *001* (Figure 9-e). Then *001* responds to *011* with a *ping* of its own (Figure 9-f). In Figure 9-g, we show the resulting state of the cube. Node *110* realizes that, after the departure of *111*, it has the highest logical address, assumes that it is the *HRoot*, and starts sending *beacons*. Nodes *000* and *001* realize that they are missing a node (actually, each other) in their respective neighborhood tables, and also sent *beacon* messages. Eventually, nodes *000* and *001* pick up each others' *beacon*s and send *ping*s to each other (Figure 9-h). This completes the repair procedure, and the hypercube has returned to a stable state.

5 Verification and Implementation

We used the **Spin** protocol verification tool [12] to aid in the development of the *HyperCast* Protocol. *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. To verify the *HyperCast* design in *Spin*, the entire *HyperCast* protocol specification, as well as a system for simulating multiple hypercube nodes was encoded using the Process Meta Language (*PROMELA*). In addition to checking for deadlocks and non-progress cycles, *Spin* was used to ensure that every execution path resulted in a stable hypercube.

Due to the unavoidable state space explosion when using a tool such as *Spin*, we are only able to analyze hypercubes with at most 6 nodes. While verification cannot be used to prove results for large hypercube sizes, we assert that, for the purposes of verification, there is little qualitative difference between a hypercube of six nodes and a hypercube of several thousand nodes. It is unlikely that non-progress cycles and deadlocks exist in large hypercubes, which *do* not have analogous fault modes in a 6-node hypercube. We wish to emphasize, however, that our verification with *Spin* is not equivalent to a formal verification of the protocol.

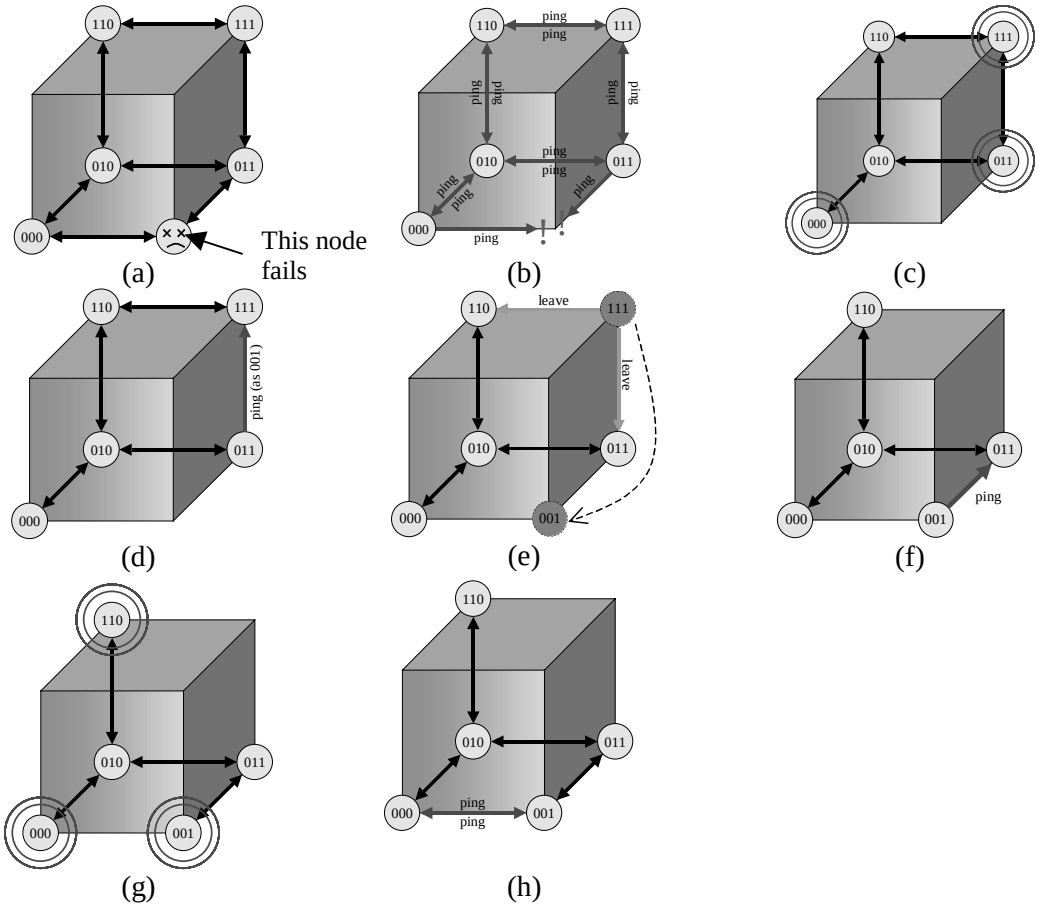


Figure 9: Repairing a tear.

The *HyperCast* protocol was implemented using the **Java** programming language. The total size of the implementation is about 5,000 lines of code. Java was chosen for its portability to multiple platforms and its easy-to-use threading constructs [4]. The implementation was an exact port of the code written in PROMELA. Two sockets are used for each hypercube node, one for unicast packets, and one for multicast packets on the control channel.

6 HyperCast Experimental Validation

To determine the scalability properties of the *HyperCast* protocol, we have tested the Java implementation in a testbed environment. The protocol testbed used is 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 a 533 MHz DEC Alpha with 256 MB of RAM running Linux 2.0.35 [15]. The Centurion cluster machines are connected with a 100 Mbit/s switched Ethernet network. Up to 32 logical hypercube nodes are run on a single machine.[5]

The goal of the experiments is to find answers to the following questions.

What is the overhead of the protocol, and how does the overhead scale with increased size of the hypercube? The overhead of the protocol consists of the (unicast and multicast) control messages *ping*,

beacon, *kill*, *leave*. Of particular importance for scalability is that the volume of *beacon* messages be low. Note that, in the current implementation, *beacon* messages are sent to all nodes of the hypercube via IP multicast.

How much time does the protocol require to return a hypercube to a stable state? To assert scalability, the time needed to return the hypercube to a stable state should not depend on the size of the hypercube. The time to reconstitute stability gives an indication to how quickly the *HyperCast* protocol can adapt to dynamic changes in the group membership.

We attempt to answer these questions in a set of experiments where we add or delete a large number of nodes, and measure the time and the amount of control traffic transmitted until the hypercube reaches a steady state.[6] The performance measures in the experiments are as follows:

- 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. (Time is measured in multiples of $t_{\text{heartbeat}}$.)

6.1 Experiment 1: Multiple Nodes Joining the Hypercube

In Experiment 1, we examine a scenario where multiple nodes want to simultaneously join the hypercube. We measure the time until the *HyperCast* protocol establishes a stable hypercube. In the experiment, we vary the number N of nodes that are already present in the hypercube ($N = 2^{i/2}$, where i ranges from 0 to 18), and the number J of nodes which want to join the hypercube when the experiment begins ($J = 2^{i/2}$, where i ranges from 0 to 16).

At the start of each experiment, there is a stable hypercube with N nodes, and J nodes want to join the multicast node (All J nodes are in state *Joining*). An experiment is completed when the hypercube contains $N + J$ nodes and is in a stable state. We measure the time until stability is reached, as well as the unicast and multicast traffic which is transmitted over the duration of the run. Figure 10(a) shows for all *(N, J)* pairs, the time until the hypercube stabilizes. Note that the plotted graph is constant as a function of N. Also, the plot indicates that there is little correlation between the number of nodes present in the hypercube and the time to attain a stable hypercube. The increase in time with respect to the number of joining nodes J (on the logarithmic axis) indicates a linear relation between the number of joining nodes and the time needed. This behavior is expected, since the process of adding one node to the hypercube should take a constant amount of time.

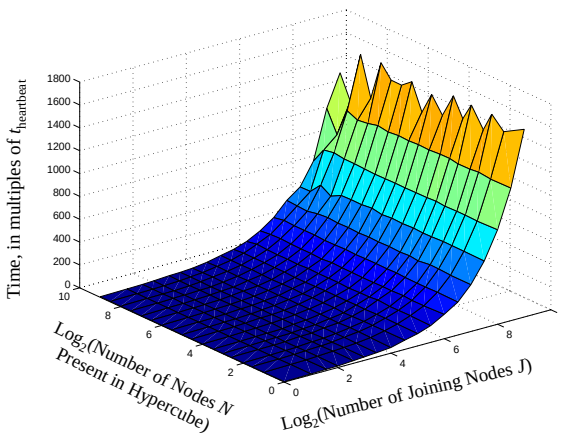
Figure 10(b) shows the average number of unicast packets sent or received by a station per $t_{hearbeat}$ time units. The values are averaged over the entire duration of the experiment. The data indicates that the unicast traffic at a node grows on a logarithmic scale. Since unicast transmissions are primarily *ping* messages between neighbors, the behavior is as expected, as the average number of neighbors of a node is approximately equal to the dimension of the hypercube, which is close to $\log_2(N + J)$.

Figure 10(c) 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 strong correlation between multicast traffic and the number of nodes present in the hypercube. There is, however, a correlation between the multicast traffic and the number of nodes joining the hypercube. This correlation is due to the *beacons* sent by newly joining nodes.

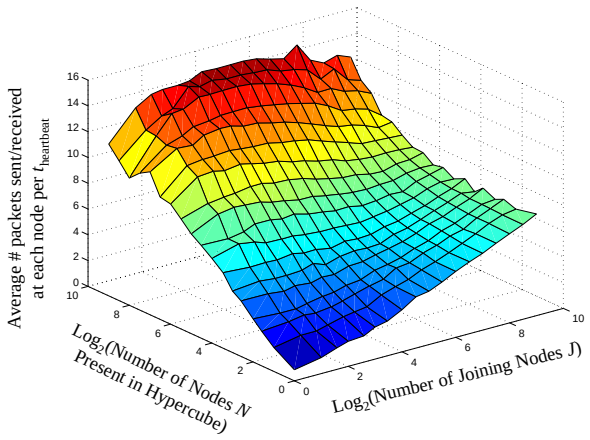
---

[5] IPC processing in the Java Virtual Machine (JVM) is the bottleneck when running multiple hypercube nodes on a single machine. The limit of 32 nodes per machine is a result of the restriction on the maximum number of sockets that can be handled by the JVM.

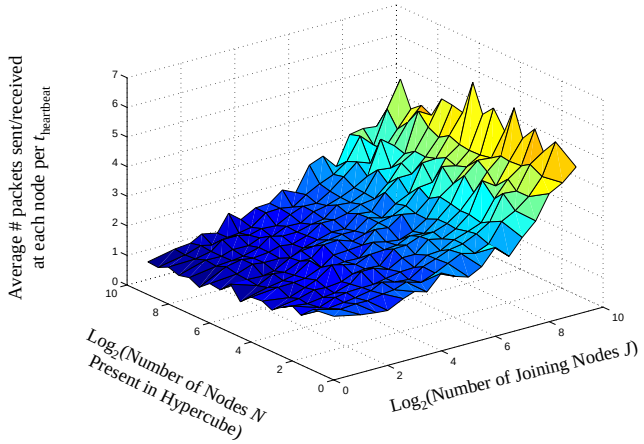
[6] We refer to [20] for additional experiments, and for a more extensive description of the experimental setup.



(a) Time until the Hypercube reaches a stable state.



(b) Average number of unicast packets sent and received per node and per time unit over the duration of the experiment.



(c) Average number of multicast packets sent and received per node and per time unit over the duration of the experiment.

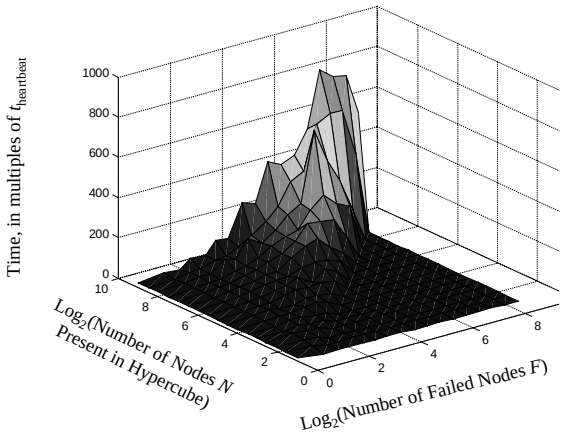
Figure 10: Results from Experiment 1.

Overall, Experiment 1 shows that the process of adding nodes to the hypercube scales well to larger group sizes. For applications, which require low latency in join operations can use a lower value of $t_{heartbeat}$, thereby reducing the time needed to add a node to the hypercube.

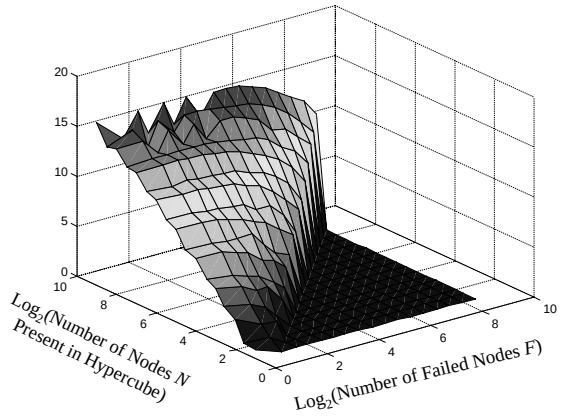
6.2 Experiment 2: Multiple Node Failures

This experiment examines a scenario where multiple nodes in a stable hypercube fail simultaneously. The setup is similar to that in Experiment 1. At the beginning of the experiment, N nodes are present in the hypercube ($N = 2^{i/2}$, where i ranges from 0 to 18), and immediately after the experiment is started, F nodes fail ($F = 2^{i/2}$, where i ranges from 0 to 16). The experiment, for a pair of values (N, F) terminates when the hypercube with $N - F$ node has reached a stable state. As before, we measure the time until stability is reached, as well as the unicast and multicast traffic over the duration of the experiment.

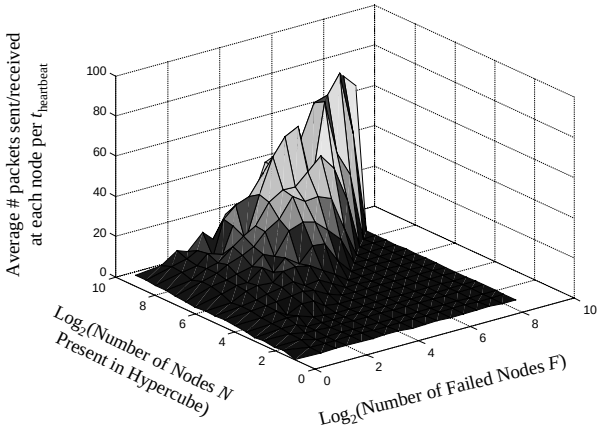
Figure 11(a) shows the relationship between the time needed to reach achieve stability of the hypercube and the parameters N and F. (Note that pairs (N, F) with $F > N$ are infeasible points). The graph shows that the time to repair a cube does not increase significantly with N, when F is small.



(a) Time until the Hypercube reaches a stable state.



(b) Average number of unicast packets sent and received per node and per time unit over the duration of the experiment.



(c) Average number of multicast packets sent and received per node and per time unit over the duration of the experiment.

Figure 11: Results from Experiment 2.

However, for large number of failures, the time to repair the cube grows linearly with the number of failed nodes.

Note that, as F approaches values close to N, the time to repair the hypercube decreases. This is explained as follows. If the number of failed nodes is close to the number of nodes in the initial hypercube, the hypercube is rather small, hence, requiring less time to reach a stable state.

Figure 11(b) shows the average rate of unicast packet transmissions during the repair operations. Since unicast transmissions are primarily *ping* transmissions, the number of transmitted *pings* is proportional to the 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 $\log_2(N - F)$. The data in Figure 11(b) confirms this relation.

Figure 11(c) 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*.

7 Conclusions

We are pursuing a novel approach to the problem of scalable multicast in packet-switched networks. The key to our approach is to organize members of a multicast group in a logical *n-dimensional hypercube*. By exploiting the symmetry properties in a hypercube, operations that require an exchange of feedback information between multicast group members can be efficiently implemented.

In this paper, we presented the design, specification, verification, and evaluation of the *HyperCast* protocol, which maintains members of a dynamically changing multicast group in a logical hypercube topology. The implementation has been tested for group sizes of up to 1024 nodes. The data indicates that larger group sizes may be reached.

The *HyperCast* protocol organizes nodes into a hypercube topology and has been tested thoroughly with that goal in mind. However, at present, the hypercube protocol is not supporting any applications. The next step of our work is to build protocol mechanisms which use the symmetrical hypercube topology to support applications. For example, we plan to use the tree embedding algorithm from Section 3 to provide ARQ error control.

References

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

[2] S. Armstrong, A. Freier, and K. Marzullo. Multicast Transport Protocol. *Request for Comments* RFC 1301, Internet Engineering Task Force, February 1992.

[3] J. Bolot. End-to-End Packet Delay and Loss Behavior in the Internet. *In: Proc. ACM Sigcomm '93*, 23(4):289-298, September 1993.

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

[5] J.M. Chang and N.F. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computing Systems*, 2(3):251-273, August 1984.

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

[7] J. Crowcroft and K. Paliwoda. A Multicast Transport Protocol. *In Proc. ACM Sigcomm '88*, pages 247-256, August 1988.

[8] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint Communications: A Survey of Protocols, Functions, and Mechanisms. *IEEE Journal on Selected Areas in Communications*. Special Issue for Multipoint Communications, 15(3): 277- 290, April 1997.

[9] S. Floyd, V. Jacobson, S. McCanne, C.G. Liu, and L. Zhang. A Reliable Framework for Light-Weight Sessions and Application Level Framing. *In: Proc. ACM Sigcomm'95*, August 1995.

[10] A. Frier and K. Marzullo. MTP: An Atomic Multicast Transport Protocol. *Technical Report*, Cornell University, 1990.

[11] H. W. Holbrook, S.K. Singhal, and D.E. Cheriton. Log-based Receiver-Reliable Multicast for Distributed Interactive Simulation. *In: Proc. ACM Sigcomm '95*, August 1995.

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

[13] M. G. W. Jones, S. A. Sorensen, and S. Wilbur. Protocol Design for Large Group Multicasting: The Message Distribution Protocol. *Computer Communications*, 14(5):287--297, 1991.

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

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

[16] B.N. Levine, D.B. Lavo and J.J. Garcia-Luna-Aceves. The Case for Reliable Concurrent Multicasting Using Shared Ack Trees. In: *Proc. ACM Multimedia '96*, November 1996.

[17] B. N. Levine and R. Rom. Supporting Reliable Concast with ATM Networks. *Technical Report*, Sun Research Labs SDS-96-0517, January 1997.

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

[19] J. Liebeherr and B. S. Sethi. A Scalable Control Topology for Multicast Communications. *In: Proc. IEEE Infocom `98*, March 1998.

[20] J. Liebeherr and T.K. Beam, HyperCast Protocol: Design and Evaluation, *Technical Report*, University of Virginia, July 1999 *(in preparation).*

[21] C. K. Miller, *Multicast Networking and Applications*, Addison-Wesley, 1998

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

[23] C. Papadopoulus, G. Parulkar, and G. Varghese. An Error Control Scheme for large-Scale Multicast Applications. 1997.

[24] S. Paul, K.K. Sabnani, J.C.-H. Lin, and S.Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *IEEE Journal on Selected Areas in Communications. Special Issue for Multipoint Communications*, 15(3):407 - 421, April 1997.

[25] S. Paul, *Multicasting on the Internet and Its Applications*, Kluwer Academic Publishers, 1998.

[26] M. Pullen, M. Myjak, and C. Bouwens. Limitations of Internet Protocol Suite for Distributed Simulation in the Large Multicast Environment. *IETF Internet-Draft*, March 1997.

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

[28] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *Request For Comments* RFC 1889, Internet Engineering Task Force, January 1996.

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

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

[31] B. Whetten, S. Kaplan, and T. Montgomery. A High Performance Totally Ordered Multicast Protocol. *In: Proc. IEEE Infocom '95*, 1995.

[32] X. R. Xu, A. C. Myers, H. Zhang, and R. Yavatkar. Resilient Multicast Support for Continuous-Media Applications. *In: Proc. NOSSDAV 1997*, 1997.

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