

# Scalable Stability Detection Using Logical Hypercube\*

Roy Friedman<sup>†</sup>    Shiri Manor<sup>‡</sup>  
Department of Computer Science  
Technion - Israel Institute of Technology  
Haifa 32000  
ISRAEL  
{roy,smanor}@cs.technion.ac.il

Katherine Guo  
Bell Laboratories  
Networking Research Lab  
101 Crawfords Corner Road  
Holmdel, New Jersey 07733  
kguo@bell-labs.com

February 26, 2002

## Abstract

This paper proposes to use a *logical* hypercube structure for detecting *message stability* in distributed systems. In particular, a stability detection protocol that uses such a superimposed logical structure is presented, and its scalability is being compared with other known stability detection protocols. The main benefits of the logical hypercube approach are scalability, fault-tolerance, and refraining from overloading a single node or link in the system. These benefits become evident both by an analytical comparison and by simulations. Another important feature of the logical hypercube approach is that the performance of the protocol is in general not sensitive to the topology of the underlying physical network.

**Keywords:** Distributed Systems, Reliable Multicast, Group Communication, Scalability.

---

\*A shorter, preliminary version of this work appeared in SRDS '99

<sup>†</sup>Partially supported by the Israeli Ministry of Science, Basic Infrastructure Fund, Project #9762.

<sup>‡</sup>Now at Intel Israel

# 1 Introduction

*Reliable multicast* has been recognized as a key feature in many distributed systems, as it allows reliable dissemination of the same message to a large number of recipients. Consequently, reliable multicast is supported by many middlewares such as *group communication* (Isis [5], Horus [35], Transis [10], Ensemble [1], Relacs [3], Phoenix [23], and Totem [26], to name a few), protocols like RMTP [28] and SRM [12], and in the near future standards like CORBA [27].

Reliable multicast typically involves storing copies of each message either by several dedicated servers, or as is typically done in group communication toolkits, by all nodes in the system. In order to limit the size of buffers, systems and middlewares supporting reliable multicast employ a *stability detection* protocol. That is, such systems and middlewares must detect when a message has been received by all of its recipients, or in other words has become *stable*, at which point it can be discarded.

Stability detection protocols must balance the tradeoff between how fast they can detect that a message is stable once it has become stable, and the overhead imposed by the protocol. On one hand, the faster the protocol can detect stability, the smaller the buffers need to be. On the other hand, if the stability protocol generates too many messages, the overhead imposed on the system will be prohibitively high. Hence, the performance and scalability of the stability detection protocol affects the overall scalability of reliable multicast.

This paper proposes to structure stability detection protocols by superimposing a *logical hypercube* [17, 21] on the system. That is, in our protocol, messages generated by the stability detection protocol only travel along logical hypercube connections, regardless of the underlying physical network topology. We claim that this logical hypercube approach has several appealing properties ( $n$  below refers to the number of nodes in the system):

**Scalability:** Each node needs to communicate with only  $\log(n)$  nodes.

**Performance:** The logical hypercube structure guarantees that the number of hops the stability information need to travel is at most  $\log(n)$ .

**Fault-Tolerance:** Hypercubes offer  $\log(n)$  node distinct paths between every two nodes,

therefore it can sustain up to  $\log(n)$  failures.

**Regularity:** Hypercubes have a very regular structure, and in our protocol every node plays exactly the same role. Thus, no node is more loaded than others. Also, code regularity tends to decrease the potential for software bugs in protocol implementation.

In this paper, we explore the performance and scalability of our proposed protocol, and compare it with other known stability detection protocols, namely, a fully distributed protocol, a coordinator based protocol, and a tree-based protocol [13, 15]. The comparison is done both analytically and by simulations. Our results show that the logical hypercube based protocol compares favorably with other protocols we have investigated, and confirm our assumptions about the use of logical hypercubes, as mentioned above.

During our simulations we have discovered another interesting property, which is shared by both logical hypercube based protocols and tree based protocols: Our measurements are carried over randomly generated network topologies, and no attempt is made to match the underlying physical topology to the logical flow of the protocol. Yet, both the tree based protocol and the hypercube based protocol appear to be *insensitive* to the network topology, giving consistent results regardless of the topology. We believe that this is also an important aspect of hypercube based protocols (and tree based protocols), since in practical distributed systems, the underlying network topology is rarely known, and might change as either the system or the network changes or both of them evolve.

## 1.1 Related Work

Many group communication toolkits, for example, ISIS [5], Horus [35], and Ensemble [16], employ a fully distributed protocol, along the lines of the *FullDist* protocol we present in Section 2. As we discuss later in Section 3, this protocol is not very scalable.

Guo et al investigated the scalability of a variety of stability detection protocols in [13, 14, 15]. These include, for example, the fully distributed protocol *FullDist*, the coordinator based protocol we refer to as *Coord*, and a tree based protocol to which we refer in this work as *S\_Coord*, all discussed in more detail in Section 2. Guo’s work was also done primarily using simulations. However, in [15] it is assumed that the physical topology of the network matches

the logical structure of the protocol, while in both [13] and in our work, we do not make this assumption. This difference is important, since many distributed applications do not control the underlying network topology, nor have access to the routers. Thus, investigating the behavior of the protocol when there is no correspondence to the actual network topology is of great interest.

Previous works on both unreliable and reliable multicast have suggested the use of a logical ring as a form of improving performance when running on a shared bus communication medium. These works include the Totem project [26] and the work of Cristian and Mishra [9].<sup>1</sup> Rings are useful in avoiding collisions, and offer moderate scalability since each node only communicates with two additional nodes. However, the scalability of rings is limited too, since information must traverse the entire ring in order to disseminate from one node to every other node [13].

Hypercubes were originally proposed as an efficient interconnect for massively parallel processors (MPP) [17, 21]. A great body of research has been done in solving parallel problems on hypercubes, as described in [17, 21]. In particular, much work has been done in the area of routing, one-to-all and all-to-all communication and gossiping in hypercubes [4, 7, 8, 11, 19, 31].

The HyperCast protocol maintains a logical hypercube structure among group members, such that heartbeats and application messages are sent along the arcs of a logical hypercube [22]. The HyperCast work was carried independently and concurrently to the conference version of this paper, and did not address stability issues. Also, with the HyperCast protocol, some nodes in an incomplete hypercube have smaller degree than others, which reduces its fault-tolerance. For example, when the hypercube has  $2^n + 1$  nodes, one of the nodes has only a single neighbor according to HyperCast. In contrast, our construction guarantees that even for incomplete hypercubes, the degree of each node and the number of node independent paths between each pair of nodes is roughly  $\log(n)$ .

Recently there have been several bodies of work on generating an approximation of hy-

---

<sup>1</sup>In fact, a logical ring is used for communication over a shared access medium in the IEEE 802.4 standard [33], known also as *token bus*. However, IEEE 802.4 does not guarantee reliable multicast, and the use of a logical ring is done to improve the throughput by avoiding collisions.

percube topologies in a fully distributed manner, and providing efficient routing and lookup services in these overlay networks [29, 32, 36]. These systems can be used, for example, as an infrastructure for large scale publish/subscribe systems.

## 2 Stability Detection Protocols

As discussed in Section 1, stability detection protocols aim to detect when messages become stable, that is, they have been received by all of their intended recipients, so their copies can be discarded. In this section we describe three existing stability detection protocols, to which we later compare our logical hypercube based protocol in terms of performance, scalability, and fault tolerance. In all cases we assume a fixed set of  $n$  nodes, known in advance, and numbered from 0 to  $n - 1$ . We also assume that the communication channels preserve FIFO ordering, and that there is an underlying mechanism guaranteeing reliable point-to-point message delivery.<sup>2</sup>

In order to present the stability detection protocols, we introduce the following notation: We denote by *ArrayMin* the element-wise array minimum of  $n$  given arrays. That is, given  $n$  arrays of length  $k$ ,  $R_1, \dots, R_n$ ,  $S = \text{ArrayMin}(R_1, \dots, R_n)$  implies that for each  $j \in [0, k - 1]$ ,  $S[j] = \min(R_1[j], \dots, R_n[j])$ .

All protocols proceed in *rounds*, executed sequentially. In each round of the protocol, the nodes attempt to establish the stability of messages that were received prior to that round (although if some messages become stable during a round, they might be detected by that round as well). Once a round ends, the next round does not start until  $\Delta$  time units have passed;  $\Delta$  is some integer which can be adjusted by the system administrator. The optimal value of  $\Delta$  depends on message sending rate and buffer size of each node, as the goal of a stability detection protocol is to release stable messages from buffers before buffer overflow happens [13]. As will become evident shortly, we are interested in the latency of each round. We are now ready to present the protocols.

---

<sup>2</sup>Note that reliable point-to-point message delivery does not guarantee reliable multicast, since it allows a situation in which a node fails and some of its messages were delivered by some processes, but not by all of them.

## 2.1 *Coord* Protocol

*Coord* is a coordinator based protocol. That is, each node  $i$  maintains an  $n$ -element array  $R_i$  whose  $j$ -th entry  $R_i[j]$  is the sequence number of the last message received by node  $i$  from node  $j$ . One of the nodes serves as the *coordinator*. The coordinator multicasts a **start** message. Each node that receives the **start** message replies with a point-to-point **ack** message to the coordinator. The **ack** message of node  $i$  contains array  $R_i$ . After receiving **ack** messages from all nodes, the coordinator constructs the minimum array  $S$  using the function *ArrayMin* described before. Following this, array  $S$  contains the sequence number for the last stable message sent from each node. The coordinator then multicasts an **info** message containing the array  $S$ . The pseudo code is given in Figure 1. Note that in the pseudo code, Line 1 and Line 8 start a protocol round at the coordinator and non-coordinator nodes, respectively.

## 2.2 *FullDist* Protocol

*FullDist* is a fully distributed protocol. That is, each node  $i$  keeps a stability matrix of size  $n \times n$  in which  $M_i[k, j]$  is the sequence number of the last message  $i$  knows about that was received by node  $k$  from node  $j$ .  $M_i[i, j]$  is the sequence number of the last message received by node  $i$  from node  $j$ . The minimum of the  $j$ -th column across the nodes represents the sequence number of the last message sent by node  $j$  and has been received by every node. At the beginning of each round, the first node<sup>3</sup> multicasts an **info** message, which contains the first row of its matrix  $M_0$ . Each node  $i$  that receives the **info** message replies with a multicast of an **info** message. The **info** message contains row  $i$  of its matrix  $M_i$ . Every node  $i$  replaces the  $k$ -th row of its matrix  $M_i$  with the row it received in the **info** message from node  $k$ . The pseudo code is presented in Figure 2. Note that in the pseudo code, Line 1 and Line 3 start a protocol round at node 0 and other nodes, respectively.

---

<sup>3</sup>There is no significance in the choice of the first node. The first node that starts that protocol can be chosen in any deterministic way, for example, according to node ids.

**Notation**

each node  $i$  maintains the following arrays:

$R_i$  – sequence number array

$S_i$  – stability array

$ArrayMin$  – element-wise minimum of the input arrays

**Initialization of node  $i$** 

$S_i := [0, 0, 0, \dots, 0]$ ;

*< At coordinator  $j$  >*

**Step 1**

1: multicast(**start**);

**Step 2**

2: wait until receive(**ack**,  $R_i$ ) from all nodes;

3:  $S_j := ArrayMin(R_1, R_2, \dots, R_n)$ ;

**Step 3**

4: multicast(**info**,  $S_j$ );

5: label all messages received from every node  $k$  with sequence number  $P \leq S_j[k]$  as stable;

6: wait( $\Delta$ );

7: goto **Step 1**;

*< At non-coordinator  $i$  >*

**Step 1**

8: wait until receive(**start**) from *coordinator*;

**Step 2**

9: send (**ack**,  $R_i$ ) to *coordinator*;

**Step 3**

10: wait until receive(**info**,  $S$ ) from *coordinator*;

11:  $S_i := S$ ;

12: label all messages received from every node  $k$  with sequence number  $P \leq S_i[k]$  as stable;

13: goto **Step 1**;

Figure 1: *Coord* protocol.

**Notation**

$\min(R)$  – the minimum value in array  $R$

$\text{row}(M_i, j)$  – the  $j$ -th row of matrix  $M_i$

$\text{col}(M_i, j)$  – the  $j$ -th column of matrix  $M_i$

each node  $i$  maintains the following:

$M_i$  – sequence number matrix

$\text{receive\_from}_i$  – the number of **info** messages node  $i$  has received in the current round

**Initialization of node  $i$** 

$\text{receive\_from}_i := 0;$

$\forall k, j, \text{ where } k \neq i \ M_i[k, j] := 0;$

**Step 1**

< At node 0 >

1: wait( $\Delta$ );

2: multicast(**info**,  $\text{row}(M_0, 0)$ );

< At node  $k \neq 0$  >

3: upon receiving (**info**,  $R$ ) from node 0 do

4:      $\text{row}(M_k, 0) := R;$

5:      $\text{receive\_from}_k := \text{receive\_from}_k + 1;$

6:     multicast(**info**,  $\text{row}(M_k, k)$ );

7: done;

**Step 2**

< At every node  $k$  >

8: upon receiving (**info**,  $R$ ) from node  $i$  do

9:      $\text{row}(M_k, i) := R;$

10:      $\text{receive\_from}_k := \text{receive\_from}_k + 1;$

11:     if ( $\text{received\_from}_k = n$ ) then

12:         label all messages received from every node  $j$  with sequence number  $P \leq \min(\text{col}(M_k, j))$  as stable;

13:          $\text{receive\_from}_k := 0;$

14:         goto **Step 1**;

15:     endif;

16: done;

Figure 2: *FullDist* protocol.

## 2.3 *S\_Coord* Protocol

Several tree-structured protocols were introduced in [15]. In this work we take *S\_Coord* as representative of tree structured protocols, since it appeared to perform the best in [15]. In *S\_Coord*, a logical tree is superimposed on the network. Each node  $i$  maintains an  $n$ -element array  $R_i$  whose  $j$ -th entry  $R_i[j]$  is the sequence number of the last message received by node  $i$  from node  $j$ . The root starts the protocol by multicasting a **start** message. The leaves send **ack** messages to their parents, containing array  $R_i$ . Each node  $i$  that receives an **ack** message from one of its children, calculates the minimum of the array it has received and its own array  $R_i$ , and stores the result in array  $M_i$ . After receiving **ack** messages from all its children in the tree, internal node  $i$  sends  $M_i$  to its parent. The root stores  $M_i$  in array  $S_i$  and multicasts an **info** message containing array  $S_i$ . Each node  $i$  that receives an **info** message containing array  $S$ , sets  $S_i$  to  $S$ . Array  $S_i$  contains the sequence number of the last stable message sent from each node. See Figure 3 for pseudo code of the protocol. Note that in the pseudo code, Line 1 and Line 2 start a protocol round.

# 3 Logical Hypercube Based Stability Detection

## 3.1 Hypercubes

An  $m$ -cube is an undirected graph consisting of  $2^m$  vertices, labeled from 0 to  $2^m - 1$ , in which there is an edge between any two vertices if and only if the binary representation of their labels differs in one bit position. More precisely, let  $H_m$  denote an  $m$ -dimensional hypercube, which consists of  $n = 2^m$  nodes. Each node is labeled by an  $m$ -bit string,  $(X_{m-1} \dots X_k \dots X_0)$ . Bit  $X_k$  corresponds to dimension  $k$ . Two nodes  $p$  with label  $(X_{m-1} \dots X_k \dots X_0)$  and  $q$  with label  $(Y_{m-1} \dots Y_k \dots Y_0)$  are connected if and only if for some index  $j$ ,  $X_j \neq Y_j$  and  $\forall i \in \{0, m-1\}, i \neq j, X_i = Y_i$ . An example of a 4-dimensional hypercube ( $H_4$ ) is given in Figure 4.

An  $m$ -cube can be constructed recursively in the following way:

1. A 1-cube is simply 2 nodes connected by an edge. As a convention, we label the nodes 0 and 1.

### Protocol for node $i$

#### Notation

$children_i$  – the indexes of node  $i$ 's children in the tree  
 $rank_i$  – number of children node  $i$  has in the tree  
 $parent_i$  – the id of node  $i$ 's parent in tree  
 $ArrayMin$  – element-wise minimum of the input arrays

each node  $i$  maintains the following variables:

$R_i$  – sequence number array  
 $S_i$  – stability array  
 $M_i$  – an array that holds the minimum so far  
 $receive\_from_i$  – the number of **ack** messages node  $i$  has received in the current round

#### Initialization

$M_i := R_i$ ,  $S_i := [0,0,0\dots 0]$ ,  $receive\_from_i := 0$ ;

#### Step 1

< At root >

1: multicast(**start**);

#### Step 2

< At leaf  $i$  >

2: upon receiving (**start**) from root do

3:     send(**ack**, $R_i$ ) to  $parent_i$ ;

4: done;

#### Step 3

< At internal node  $i$  or root >

5: upon receiving (**ack**, $R$ ) from  $j$  do

6:      $M_i := ArrayMin(M_i, R)$ ;

7:      $receive\_from_i := receive\_from_i + 1$ ;

8:     if  $receive\_from_i = rank_i$  then

9:         if < internal node > then

10:             send(**ack**, $M_i$ ) to  $parent_i$ ;

11:         elseif < root > then

12:              $S_i := M_i$ ;

13:             multicast(**info**, $S_i$ );

14:             label all messages received from every node  $k$  with sequence number  $P \leq S_i[k]$  as stable;

15:              $receive\_from_i := 0$ ;

16:             wait( $\Delta$ );

17:             goto **Step 1**;

18:         endif;

19:     endif;

20: done;

#### Step 4

< At every non-root node  $i$  >

21: upon receiving (**info**, $S$ ) do

22:      $S_i := S$ ;

23:     label all messages received from every node  $k$  with sequence number  $P \leq S_i[k]$  as stable;

24:      $receive\_from_i := 0$ ;

25:     goto **Step 1**;

26: done;

Figure 3:  $S\_Coord$  protocol.

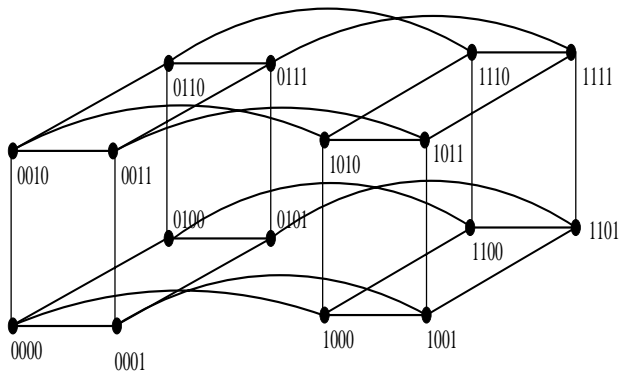


Figure 4: An example of a 4-dimensional hypercube ( $H_4$ ).

2. An  $m$ -cube is made up of two  $(m - 1)$ -cubes  $A$  and  $B$ . The labels of  $A$  are preceded by 0, and the labels of  $B$  are preceded by 1. We then add an edge between each node  $p$  in  $A$  and the node in  $B$  that only differs from  $p$  by the leftmost bit.

Hypercube is a powerful interconnection topology due to its many attractive features, as pointed out in [17, 21, 30]. These features include its regularity, a small diameter ( $\log(n)$ ), small fan-out/fan-in degree ( $\log(n)$ ), and having multiple ( $\log(n)$ ) node-disjoint paths between every two nodes.

### 3.2 The *CubeFullDist* Protocol

*CubeFullDist* is a fully distributed protocol, in the sense that every node periodically multicasts its information about message stability to its logical neighbors (in the hypercube). *CubeFullDist* employs a *gossip style* mechanism (similar to [14, 20]) to disseminate stability information. That is, in each round of the protocol, each node communicates with its logical neighbors only, until it learns what messages were received by each node at the beginning of the round. In order to save messages, we divide each round into multiple *iterations*. In each iteration  $r$ , each node sends its stability information to all its logical neighbors, and waits for a stability message from each one of them. At the end of the iteration, each node checks whether it has learned what messages were received by every other node at the beginning of the round. If it has, then the node sends its current stability information to all its neighbors and proceed to the next round. Otherwise, the node loops to the next iteration.

A more precise pseudo code for *CubeFullDist* is given in Figure 5. Each node  $i$  maintains the following variables:

- A sequence number array  $R_i$  whose  $j$ -th element  $R_i[j]$  is the sequence number of the last FIFO message received by node  $i$  from node  $j$ .
- A stability array  $S_i$  corresponding to  $i$ 's stability information at the end of each round.
- A bitmap array  $G_i$  recording the nodes from which  $i$  has learned about their stability information during this round.
- An array  $M_i$  containing the minimum sequence numbers heard so far in this protocol round.
- An integer  $r_i$ , which holds iteration number, so redundant messages belonging to previous iterations can be discarded.

A protocol round starts with Step 1. In Step 1,  $M_i$  is initialized to  $R_i$ . Node  $i$  multicasts to its hypercube neighbors a **stability** message containing  $r_i$ ,  $G_i$ , and  $M_i$ . In Step 2, node  $i$  receives messages from all its neighbors. Upon receiving a message with bitmap  $G$  and sequence number array  $M$ , node  $i$  sets its bitmap array  $G_i$  to be the bit-wise *or* of arrays  $G$  and  $G_i$ , and sets  $M_i$  to be *ArrayMin* of  $M_i$  and  $M$ . If  $G_i$  contains all 1's, this indicates that node  $i$  has heard from everyone in this round, and thus from  $i$ 's point of view the round is over. In this case, node  $i$  multicasts to its neighbors the last **stability** message in the current round, and starts a new round. Otherwise, if the round is not finished yet but  $i$  received **stability** messages from all its neighbors in this iteration, then  $i$  multicasts a **stability** message to its neighbors, and starts another iteration of Step 2.

Note that due to FIFO, and since before starting a new iteration, each node sends the last stability information known to it to all its neighbors, node  $i$  can never receive messages with  $r > r_i$ . More precisely, node  $i$  receives messages only from its neighbors. Each time one of  $i$ 's neighbors  $j$  increases its iteration number  $r_j$ , it will first multicast a message  $m_1$  containing  $G_j$  to all its neighbors where  $G_j$  contains all 1's. When node  $i$  receives  $m_1$  it increases  $r_i$ . Thus, node  $i$  increases  $r_i$  each time  $j$  increases  $r_j$ , and after  $i$  increases  $r_i$ , both

nodes  $i$  and  $j$  will have the same iteration number. Further messages from node  $j$  to node  $i$  will be delivered after  $m_1$  and after  $i$  has increased its iteration number. If node  $i$  receives a message with a lower iteration number, then this is a redundant message, probably from a neighbor that has not advanced to the current iteration by the time the message was sent, and hence the message can be ignored.

### 3.3 Incomplete Hypercubes

Hypercubes are defined for exactly  $2^m$  nodes, for any given  $m$ . However, practical systems may employ an arbitrary number of participants. A flexible version of the hypercube topology, called *incomplete hypercube* [18], eliminates the restriction on node numbers. When building logical connections in an incomplete hypercube, we strive to keep the properties that make complete hypercubes so attractive. In other words, our goals in designing the incomplete hypercubes are: (a) minimize system diameter for performance, (b) maximize the number of parallel shortest paths between two nodes in order to be fault tolerant, and (c) restrict the number of logical connections for each node to the limit of  $\log(n)$ , so the protocol remains scalable.

We denote an incomplete hypercube by  $I_m^n$ , where  $m$  and  $n$  are the dimension and the total number of nodes respectively. To achieve our goals, it is useful to note that an incomplete hypercube comprises multiple complete ones. There is a connection between node  $p$  in  $H_i$  and node  $q$  in  $H_k$  when  $k > i$  if the addresses of  $p$  and  $q$  differ in bit  $k$ . For example, consider  $I_4^{14}$  given in Figure 6. In this example there are 3 complete cubes  $H_3$  consisting of nodes  $0xxx$ ,  $H_2$  consisting of nodes  $10xx$ , and  $H_1$  consisting of nodes  $110x$ .

Our aim is to compensate for missing links in an incomplete hypercube  $I_m^n$  with respect to the complete hypercube  $H_m$ , while preserving the goals described above. This is done by adding one edge between some pairs of nodes  $p$  and  $q$  in  $I_m^n$  whose Hamming distance is 2 and are connected in  $H_m$  through a node that is missing in  $I_m^n$ . These nodes are chosen as follows: For each missing node  $z$ ,  $G_z$  is the set of nodes that  $z$  was supposed to be connected with. More formally, we denote by  $H_m \setminus I_m^n$  the set of nodes in  $H_m$  that do not appear in  $I_m^n$ . For each node  $z$  that belongs to  $H_m \setminus I_m^n$ ,  $G_z$  is defined as  $G_z = \{w | w \in I_m^n \text{ and } H(z, w) = 1\}$ .

### Protocol for node $i$

#### Notation

$neighbors_i$  – indexes of node  $i$ 's neighbors in the hypercube  
 $rank_i$  – number of neighbors node  $i$  has in the hypercube  
 $ArrayMin$  – element-wise minimum of the input arrays  
 $ArrayMax$  – element-wise maximum of the input arrays  
Each node  $i$  maintains the following variables  
 $G_i$  – gossip bitmap array  
 $R_i$  – sequence number array  
 $S_i$  – stability array at the end of each round  
 $M_i$  – minimum sequence number array in this round  
 $r_i$  – number of the current round  
 $receive\_from_i$  – the number of stability messages node  $i$  has received in the current iteration

#### Initialization

$M_i := R_i$ ;  
 $S_i := [0,0,0,\dots,0]$ ;  
 $G_i[i] := 1, \forall j \neq i, G_i[j] := 0$ ;  
 $r_i := 0$ ;  
 $receive\_from_i := 0$ ;

< At every node  $i$  >

#### Step 1

1: multicast(stability,  $G_i, M_i, r_i$ ) to  $neighbors_i$ ;

#### Step 2

2: upon receiving (stability,  $G, M, r$ ) from node  $j$  do  
3:   if ( $r = r_i$ ) then  
4:      $G_i := ArrayMax(G_i, G)$ ;  
5:      $M_i := ArrayMin(M_i, M)$ ;  
6:      $receive\_from_i = receive\_from_i + 1$ ;  
7:     if ( $G_i$  contains all 1's) then /\* start new round \*/  
8:       multicast(stability,  $G_i, M_i, r_i$ ) to  $neighbors_i$ ;  
9:        $S_i := M_i$ ;  $r_i := r_i + 1$ ;  $M_i := R_i$ ;  
10:        $G_i[i] := 1$ ;  
11:       for all  $j \neq i, G_i[j] := 0$ ;  
12:        $receive\_from_i := 0$ ;  
13:       label all messages received from node  $k$  with sequence number  $P \leq S_i[k]$  as stable;  
14:       wait( $\Delta$ );  
15:       goto Step 1;  
16:     endif;  
17:     if ( $receive\_from_i = rank_i$ ) then  
18:       /\* start another iteration \*/  
19:       multicast(stability,  $G_i, M_i, r_i$ ) to  $neighbors_i$ ;  
20:        $receive\_from_i := 0$ ;  
21:     endif;  
22: done;

Figure 5: *CubeFullDist* protocol.

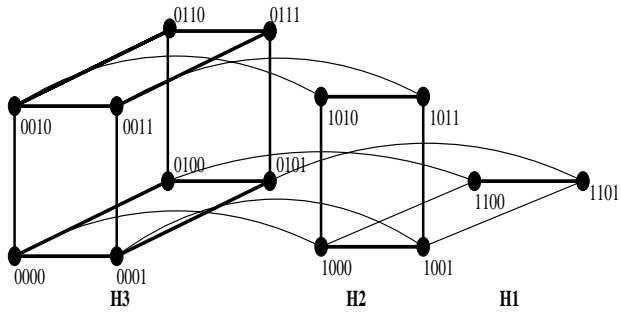


Figure 6: Incomplete hypercube with 14 nodes.

Protocol	<i>S_Coord</i>	<i>Coord</i>	<i>FullDist</i>	<i>CubeFullDist</i>
Number of protocol iterations	$O(\log(n))$	$O(1)$	$O(1)$	$O(\log(n))$
Number of messages per node per round	$O(1)$	coord $O(n)$ ; other $O(1)$	$O(n)$	$O(\log(n))$
Total number of message	$O(n)$	$O(n)$	$O(n^2)$	$O(n \cdot \log^2(n))$
Robustness to failures	no	no	yes	yes
Code regularity	no	no	yes	yes

Figure 7: Analytical comparison of the protocols.

If  $G_z$  is empty or has only one member no connection is added.  $SG_z$  is now obtained by lexicographically sorting  $G_z$  according to node ids. If  $|SG_z|$  is odd, the first node in  $SG_z$  is discarded.  $SG_z$  is now partitioned into two groups  $G_z^1$  and  $G_z^2$ .  $G_z^1$  contains the first half of ids from  $SG_z$  while  $G_z^2$  contains the second half of ids from  $SG_z$ . A connection is added between the  $i$ 'th node of  $G_z^1$  and the  $i$ 'th node of  $G_z^2$ . For example, in  $I_3^7$  node 7 is missing with respect to  $H_3$ . Thus,  $G_7 = \{3, 5, 6\}$ ,  $G_7^1 = \{5\}$ ,  $G_7^2 = \{6\}$ , and a connection is added between nodes 5 and 6.

Note that due to the way we label nodes, each node in  $H_{m-k}$  is connected in  $H_m$  to at most  $k$  nodes from  $H_m \setminus I_m^n$ . Also, we add at most one connection for each missing one. Thus, each node has a final degree between  $m$  and  $m - 1$ , and therefore the scalability and fault tolerant properties are preserved. The system diameter of  $I_m^n$  is  $m$ , and by adding links we do not increase the diameter of the system. Hence, the described incomplete hypercube matches our goals.

### 3.4 Analytical Comparison

A comparison of the four stability detection protocols is represented in Figure 7. The number of protocol iterations serves as an indication for the latency of detecting stability. In

the case of *S\_Coord*, information propagates in the tree according to the levels of the nodes, and since there are  $n$  nodes, we count this as  $\log(n)$ .

As far as scalability is concerned, *S\_Coord* (the tree based protocol) appears to be the most scalable, followed closely by *CubeFullDist*. The problem with tree protocols is that they are not fault tolerant. If one node fails, all nodes in the logical branch under it will become disconnected. In this case the tree needs to be rebuilt immediately, and the current round of the protocol is lost.

*CubeFullDist* uses more messages than *S\_Coord*. This message redundancy makes *CubeFullDist* more fault tolerant than *S\_Coord* since in *CubeFullDist* the system is logically partitioned only after  $\log(n)$  neighbors of the same node fail.

*Coord* and *FullDist* have limited scalability. In *Coord*, the coordinator receives  $O(n)$  messages from all nodes, which is infeasible when  $n$  is large. In *FullDist*, the total number of messages is  $O(n^2)$ . In contrast, in *CubeFullDist* each node receives only  $O(\log(n))$  messages, and in *S\_Coord* each node receives at most  $d$  messages, where  $d$  is the tree degree.

Looking at the number of iterations, theoretically, *CubeFullDist* is slower than *Coord* and *FullDist*. However, simulations show that actually *CubeFullDist* is much faster. The reason for this is that in *Coord*, the coordinator is the bottleneck of the protocol. In *FullDist*, the total message number is high, and each node is loaded. Loaded nodes create long message queues, which slow the overall performance, making *FullDist* and *Coord* much slower than *CubeFullDist* and *S\_Coord*.

*FullDist* and *CubeFullDist* are fault tolerant, while the other two protocols are not. Finally, *FullDist* and *CubeFullDist* have regular structure, and the same code is executed by all nodes. This property tends to yield simpler, less error prone code. On the other hand, *S\_Coord* has the advantage that it might be easier to embed a tree topology in typical physical network topologies than it is to embed a hypercube topology.

### 3.5 Weakening the Network Assumptions

We now discuss the possibility of weakening the network assumptions in each of the four stability detection protocols, to eliminate the reliable delivery requirement. Note that in

general, stability is a non-decreasing property. Since messages have sequentially increasing sequence numbers, it is generally possible to ensure the correctness of a stability detection protocol by periodically sending the most recent local information. However, to reduce network load, most of our protocols advance in rounds. In order to keep this efficiency, they may require adding a protocol round number to their messages.

Specifically, in the code of the *Coord* protocol in Figure 1, a round number must be attached to the all messages. Then, Line 1 should be repeated periodically until the coordinator receives an `ack` from all nodes, and Line 4 should be repeated periodically until the coordinator receives a new type of acknowledgment message `ack-info`. For non-coordinator members, between Lines 12 and 13 we should add sending an `ack-info` message to the coordinator. Very similar modifications are also required in the code of the *S\_Coord* protocol in Figure 3.

A round number should also be attached to all messages in the code of the *FullDist* protocol in Figure 2. Additionally, each node needs to multicast its `info` message periodically, until it receives the `info` messages of the same round from all other nodes. Also, if a node has already moved to round  $r + 1$ , and receives an `info` message from some node in round  $r$ , for some  $r$ , then the message is stale and the node in round  $r + 1$  should ignore the message. However, if a node in round  $r$  receives an `info` message in round  $r + 1$ , then it will treat the message as if it has a round number  $r$ , and process the message.

Finally, the *CubeFullDist* protocol in Figure 4 already includes round numbers, and thus it can work correctly simply by multicasting messages periodically, in a similar fashion to the other protocols. However, for efficiency reasons, the protocol also uses an internal iteration that does not advance until a node has heard from all of its logical neighbors in the current iteration. To make sure that this is indeed the case, it is advisable to add an iteration number to the `stability` messages of the protocol. Similarly, `receive_from` should only be increased when a message is received for the first time from a node in the current iteration.

In order to eliminate the requirement for FIFO delivery for stability detection protocols, it is possible to remember the last message received from each node, and check that the most recent message carries any information that is at least as recent as the previous one. If it does, the message is handled according to the protocols. Otherwise, it is ignored.

## 4 Experimental Performance

### 4.1 Simulation Model

We use the ns [25] simulator to explore the behavior of the stability detection protocols described in Sections 2 and 3. We have measured the following indices, with the goal of checking the effect of the number of nodes on these indices in the four protocols:

**RTT - Round Trip Time:** RTT of a node is the time from the beginning of a protocol round until the node recognizes that the current round of the protocol has finished. The time for detecting message stability is a function of RTT and the frequency of rounds in the stability detection protocol ( $1/\Delta$ ). If the reliable multicast protocol can deliver a message to all the receivers within  $D$  seconds, the stability detection protocol is triggered every  $\Delta$  seconds, and it can detect the message's stability within RTT seconds, then the maximum time to detect the message's stability is  $D + \Delta + \text{RTT}$  seconds. Therefore, the buffer size of unstable messages is proportional to the time it takes to detect message stability and the frequency at which stability detection messages are sent.

**Total number of messages:** This is the total number of messages sent in the system. Each message is counted only once when it is sent. The total number of messages is a good indication of processors' load.

**Hop count:** This is the total number of hops that all messages pass through. Each message is counted once at each hop that it passes on its way from source to destination. This index shows the overall protocol message overhead on all links in the system.

**Network load:** Here we measure the average network load, that is, the average number of messages on all links in the network at any given time, and the maximum queue size, that is, the maximum number of messages waiting to be sent on any of the links in the system. Note that the average network load looks at the number of messages at any given moment, while in hop count we are interested in the cumulative number of messages on all links in a full run of a protocol.

**Topology sensitivity:** This measures the difference between best result and worst result for each of the protocols on each of the indices described above. Since the logical flow of the protocol does not match the physical underlying topology, this indicates how sensitive the protocol performance is to the actual network topology.

Lastly, we have also measured the effect of node failures on the above indices on the *CubeFullDist* protocol. Note that both *S\_Coord* and *Coord* are not fault tolerant, so it is not meaningful to look at the effects of failures on them. Also, the performance of *FullDist*, as reported below, is so poor, that we decide to only show the results of failures on *CubeFullDist*.

The stability detection protocols are tested with several randomly generated network topologies. We use the random network generator GT-ITM [6] to build random network topologies with edge probability varying from 0.01 to 0.04; each protocol is run 6 times and in each run a different randomly generated topology is used. The results presented are the average of the 6 runs. The links in the simulations are chosen to be duplex 100Mbps in each direction with uniformly distributed delay between 0 to 1 ms. We assume that the total number of nodes is varied between 10 to 1900. In groups smaller than 50 nodes, all the nodes are sending messages. In larger groups, however, the number of senders is fixed at 50. The sequence number size is assumed to be 4 bytes, and thus the size of the array of stability information carried by `stability`, `ack`, and `info` messages is at most 200 bytes. Also, the message header size is set to 32 bytes, large enough for most transport protocols [2, 34].

The tree degree in *S\_Coord* is set to  $\log(n)$ , that is, each node has  $\log(n)$  children. Node 0 is chosen as the coordinator of *Coord*, and as the root of the tree in *S\_Coord*. In *CubeFullDist*, the logical hypercube is built according to node id's binary representation. Since the network is generated randomly, the fitness of logical structure to the network is random as well. In *CubeFullDist*, when the number of nodes is not power of two, the construction of incomplete hypercubes described in Section 3.3 is used.

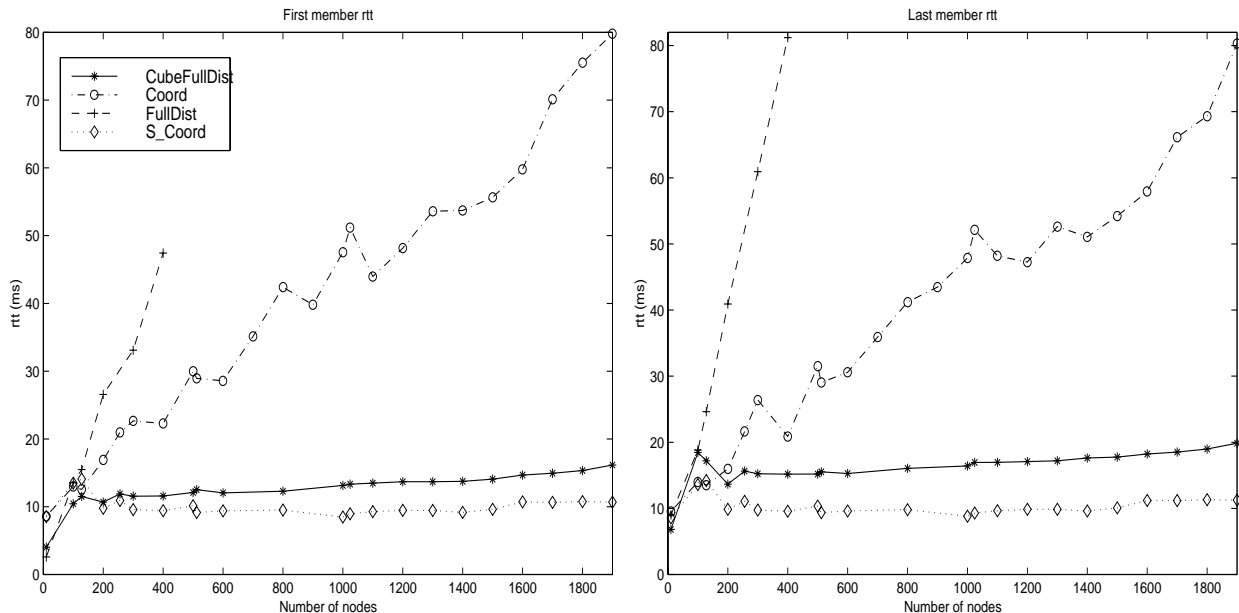


Figure 8: RTT as a function of system size.

## 4.2 Simulation Results

### 4.2.1 Round Trip Time

The RTT of the fastest and slowest nodes in detecting stability are reported in Figure 8. For both *S\_Coord* and *CubeFullDist*, RTT remains almost flat as the number of nodes increases. This is because in both protocols, all nodes are reasonably loaded, and the number of messages sent and received by each node is small. Also, it can be seen that our construction of logical incomplete hypercubes maintains its scalability goals with respect to complete hypercubes, since the graphs do not have any major jitters near and at system sizes that are power of two. Note that data points in the same set have similar RTT values: (100, 128), (200, 256, 300), (500, 512) and (1,000, 1,024).

The RTT in *Coord* increases linearly with  $n$ , since the coordinator load is proportional to  $n$ . This causes a long message queue at the coordinator and slows down the overall performance.

The RTT of *FullDist* increases dramatically as the number of nodes increases. In fact, the timing of *FullDist* is so bad that we could not check beyond 400 nodes. The reason for this is that the total load imposed by *FullDist* on the network is too high. Each node in

*FullDist* sends and receives  $O(n)$  messages, or a total of  $O(n^2)$  messages, which causes long message queues at all nodes and heavy utilization of all links and nodes in the system. (The load caused by *FullDist* can be seen in Figures 9, 10, and 11. We discuss these graphs later.)

It is interesting to notice that for both *Coord* and *S\_Coord* there is hardly any difference between first node and last node RTT. This is because the coordinator/root informs everyone after it finishes each protocol round. *FullDist*, on the other hand, shows a significant difference between first node and last node RTT when the system load is high. For 150 nodes, the RTT for the first and last node is 16ms and 19ms respectively. For 400 nodes, the RTT becomes 46ms and 82ms respectively. When the system load is low (150 nodes), the difference in RTT values come from the distributed nature of the protocol; when the system load is high, the difference comes from the fact that the system is overloaded with  $O(n^2)$  messages.

In *CubeFullDist* there is a slight difference between first node and last node RTT. This difference is caused by the distributed nature of the protocol, that is, it takes time for the information to propagate in the system.

#### 4.2.2 Network Load

The average network load and the maximum queue size measured are reported in Figures 9 and 10, respectively. For a given system size, the network load is recorded every 1 ms, and averaged over the time for each round. *FullDist* has the largest queue size. In *FullDist* each node sends and receives  $O(n)$  messages, or a total of  $O(n^2)$  messages, which results in high network load and large message queues.

In *Coord*, the maximum queue size grows linearly with the system size. This is because the coordinator receives  $O(n)$  messages. The average network load is not so high though, and in fact, is even somewhat better than *CubeFullDist*, since the total number of messages sent out in the system is  $O(n)$ .

*CubeFullDist* and *S\_Coord* have almost the same maximum queue length. The maximum queue size of *S\_Coord* is somewhat better than *CubeFullDist* since *S\_Coord* uses fewer messages. In *CubeFullDist* the average queue length and maximum queue length are very

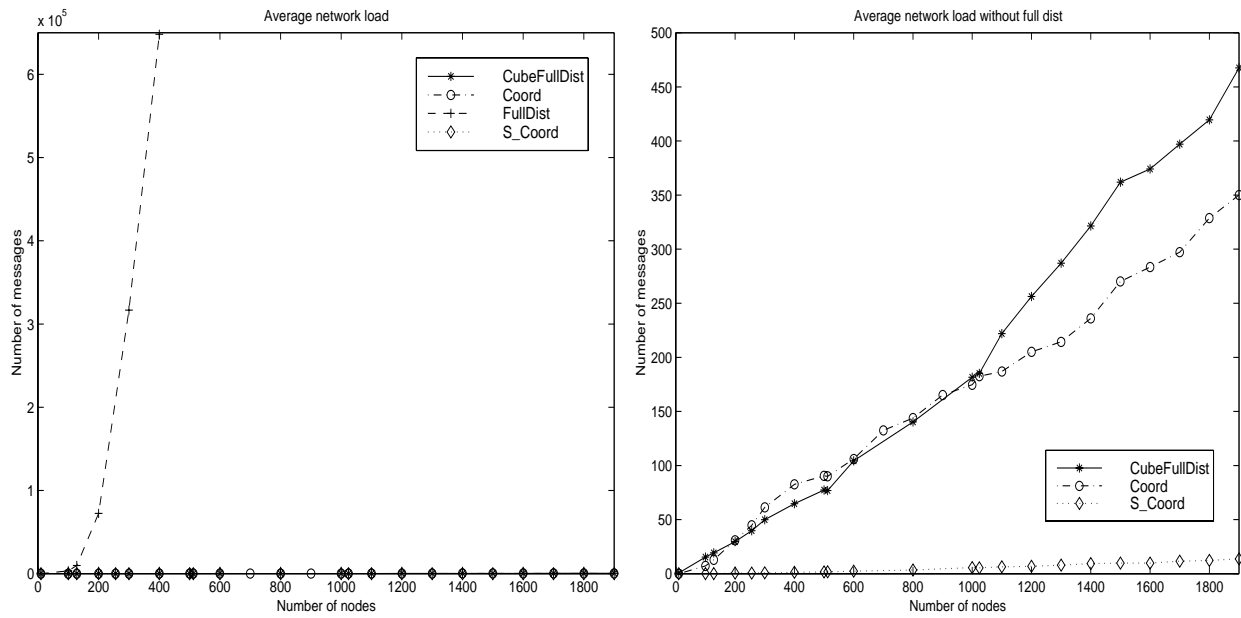


Figure 9: Average network load as a function of system size. The right graph zooms in on the results of *Coord*, *S\_Coord*, and *CubeFullDist*. (Notice difference in scale.)

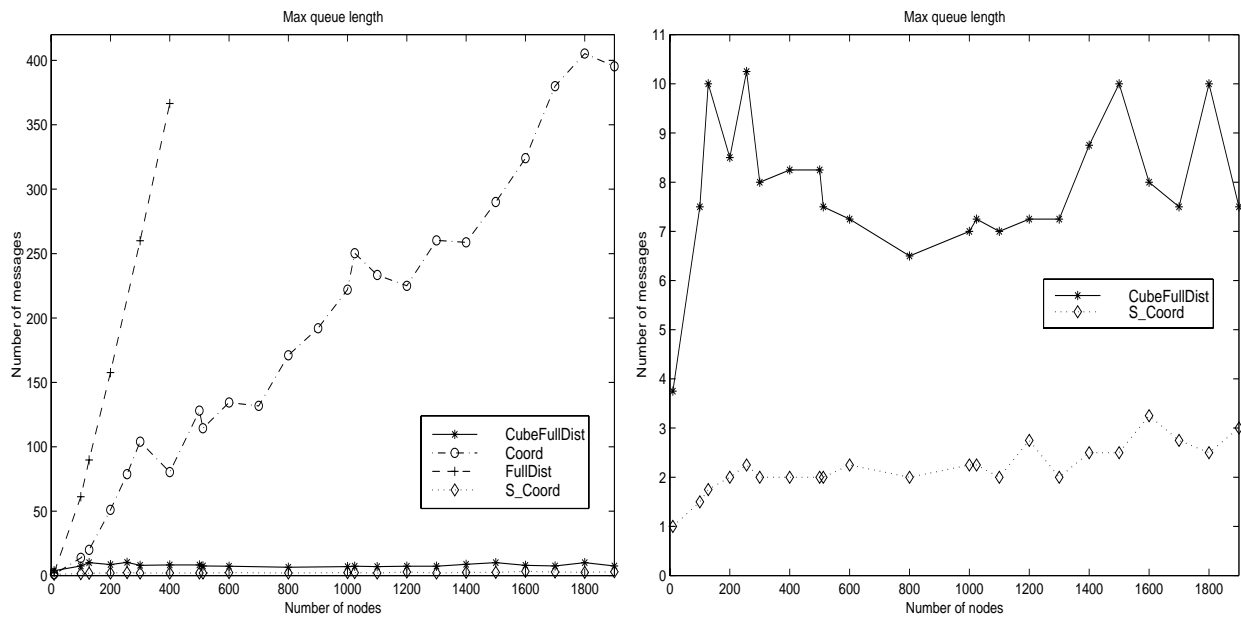


Figure 10: Maximum queue size as a function of system size. The right graph zooms in on the results of *S\_Coord* and *CubeFullDist*. (Notice difference in scale.)

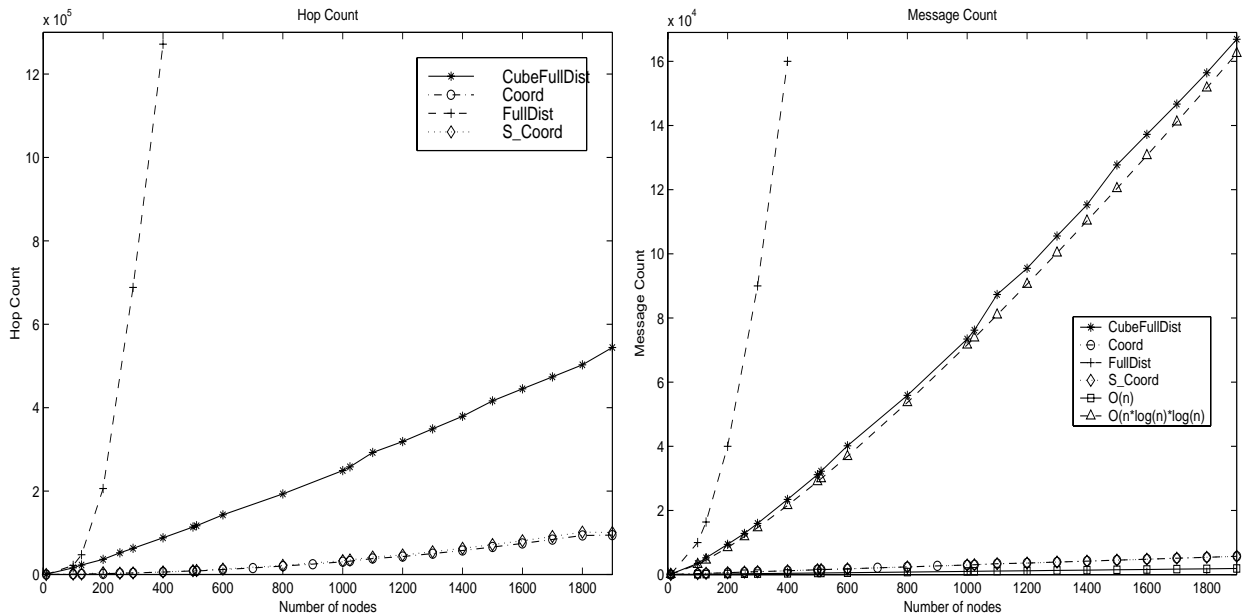


Figure 11: Hop count and number of messages as a function of system size. (Notice difference in scale.)

similar, because all nodes send and receive the same number of messages. In *S\_Coord*, on the other hand, the average queue length is very low because each node sends and receives only  $\log(n)$  messages.

### 4.2.3 Hop Count and Message Count

The total hop count and message count are reported in Figure 11. In all protocols, the theoretical analysis of message count (see Section 3.4) is reflected in the simulation graphs. *FullDist* has an enormous message count,  $O(n^2)$ , which is reflected in both graphs. *Coord* and *S\_Coord* have linear message count. In *S\_Coord*, the hop count is less than in *Coord* because most protocol messages are only sent from a node to its parent in the tree. Whereas in *Coord*, most messages are sent to the root node directly. It is not too difficult to embed a logical tree on an arbitrary physical topology to match the logical topology with the physical topology as much as possible, and therefore the actual number of hops could be even lower than the simulation result.

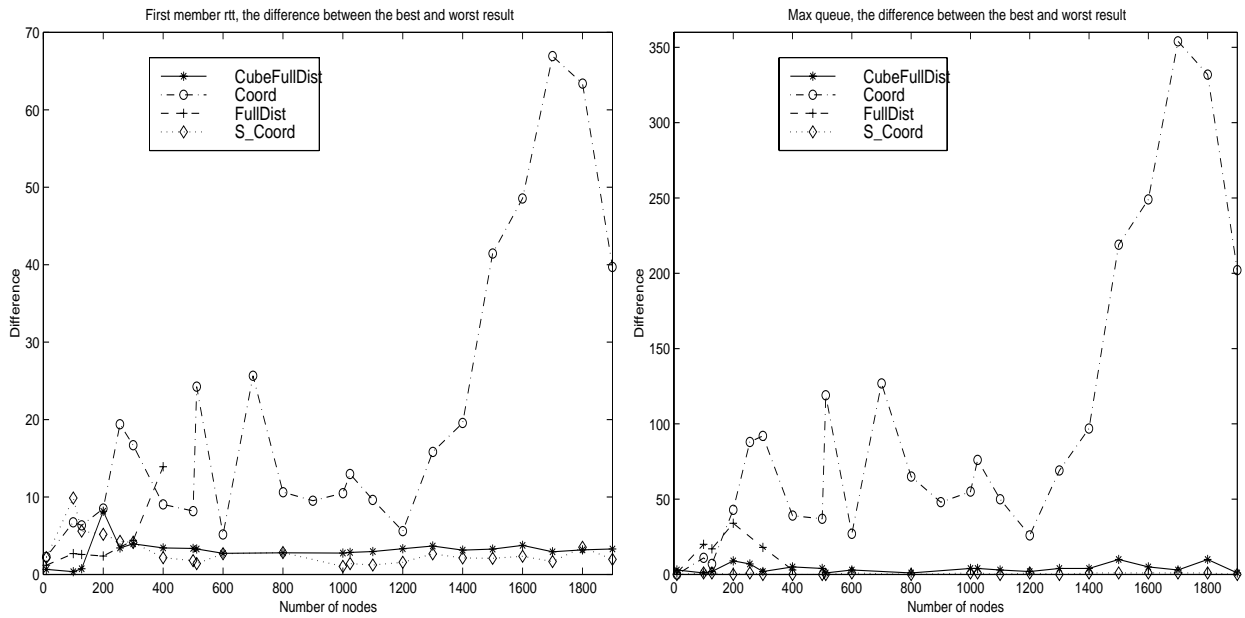


Figure 12: The difference between the worst and best result as a function of system size.

*CubeFullDist* uses  $O(n \log^2(n))$  messages, which is reflected in the message count graph. *CubeFullDist* hop count is relatively better than its message count because protocol messages are sent only to hypercube neighbors. Note that while *Coord* hop count and message count are better than *CubeFullDist*, *CubeFullDist* is faster (provides faster RTT). The reason is that in *CubeFullDist* the load is distributed in the network and in *Coord* all messages must traverse to the same root node.

#### 4.2.4 Protocol Sensitivity

The difference between the best and worst simulation result, of first RTT and max queue, are reported in Figure 12. The performance of *Coord* is very much affected by the underlying network topology. If in the randomly generated network the coordinator has many connections, *Coord* achieves quite good performance. On the other hand, if the coordinator is assigned only a few connections, these connections become bottlenecks, and the performance is very poor. The other protocols do not have such bottlenecks, so they are not very sensitive to the network topology.

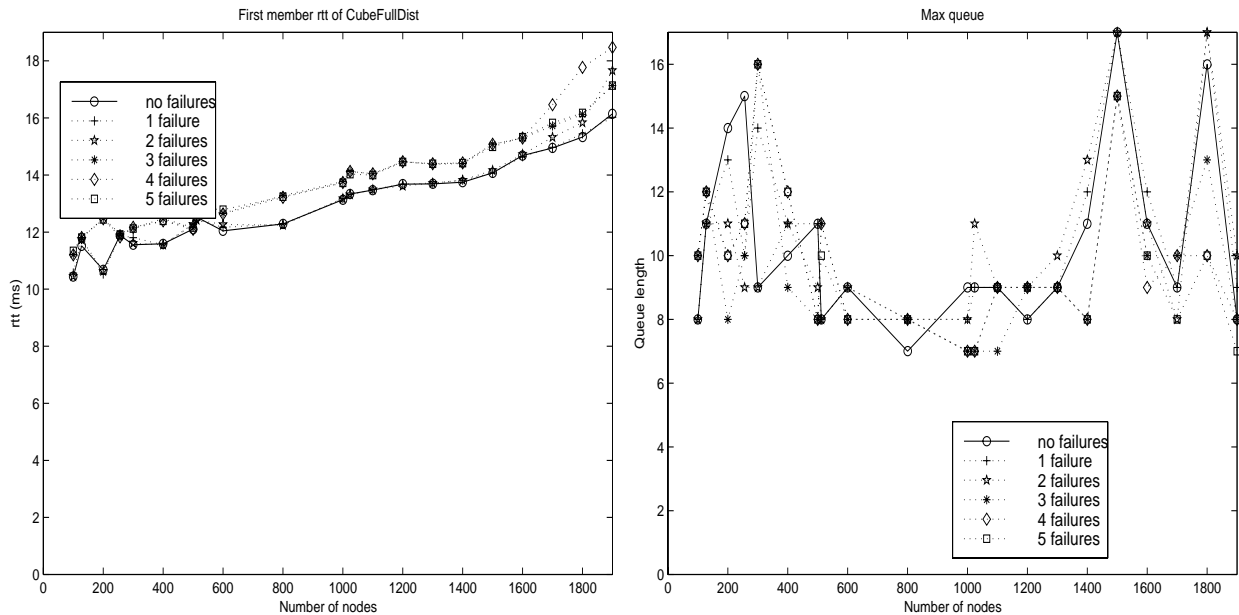


Figure 13: The effects of node failures on *CubeFullDist* performance.

#### 4.2.5 The Effect of Faults on *CubeFullDist* Performance

The effects that node faults have on *CubeFullDist* performance can be seen in Figures 13 and 14. We test the protocol with 0-5 node failures; faulty nodes are chosen randomly among node 0's neighbors. As mentioned above, *CubeFullDist* is very fault tolerant, and its performance is hardly affected by a small number of failures. In particular, note that our choice of faulty nodes is a worst case scenario, since all faulty nodes are neighbors of the same node rather than arbitrarily chosen.

## 5 Discussion and Future work

Our study indicates that superimposing a logical hypercube structure as a way of obtaining scalability and fault-tolerance, especially in the context of reliable multicast, is a promising direction. In [24], we also study the applicability of logical hypercubes to scalable failure detection and causal ordering. The applicability of logical hypercubes to other aspects of reliable multicast and group communication is still an open question.

In this work we have assumed general networks, on which the logical hypercube structure

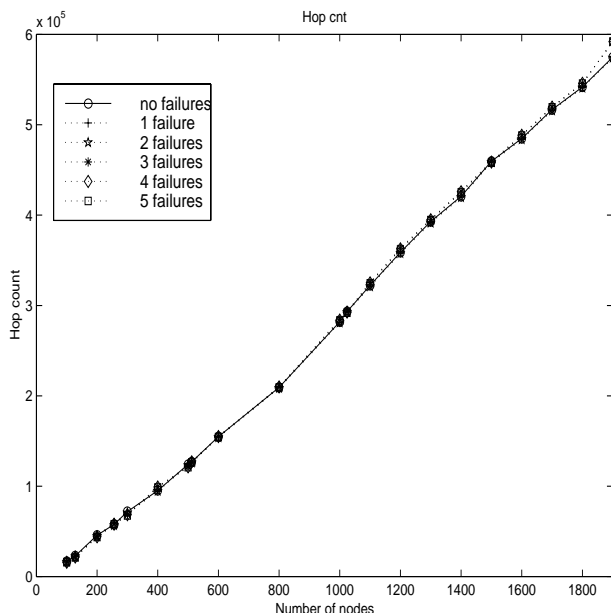


Figure 14: The effects of node failures on *CubeFullDist* performance.

(as well as the other structures) were superimposed in a random manner. Nevertheless, trying to map the logical hypercube structure to the physical network topology in a smart way might improve the performance of our protocol. Some work has already been done on matching hypercubes to other topologies, mainly trees and meshes [21], although looking at this problem in a more general context, such as the Internet, is an interesting research direction.

Finally, none of the stability detection protocols we described here assume anything about the reliable multicast protocol that it might use in conjunction with. Some optimizations, for example, piggybacking stability messages on protocol messages, might be possible by tailoring the stability detection protocol to the multicast protocol.

**Acknowledgements:** We would like to thank the anonymous reviewers for their helpful comments.

## References

- [1] The Ensemble Home Page. <http://www.cs.cornell.edu/Info/Projects/Ensemble>.
- [2] R. Ahuja, S. Keshav, and H. Saran. Design, Implementation, and Performance of a Native Mode ATM Transport Layer. *IEEE/ACM Transactions on Networking*, 4(4):502–515, August 1996.
- [3] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS-94-15, Department of Computer Science, University of Bologna, June 1994. Revised January 1995.
- [4] O. Baudon, G. Fertin, and I. Havel. Routing Permutations and 2-1 Routing Requests in the Hypercube. *Discrete Applied Mathematics*, 113(1):43–58, 2000.
- [5] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.
- [6] K. Calvert and E. Zegura. GT-ITM Random Network Generator. <http://www.cc.gatech.edu/projects/gtitm>.
- [7] B.S. Chlebus, K. Diks, and A. Pelc. Optimal Broadcasting in Faulty Hypercubes. In *Proceedings of 21st Annual International Symposium on Fault-Tolerant Computing*, pages 266–273, Montreal, Canada, June 1991.
- [8] B.S. Chlebus, K. Diks, and A. Pelc. Fast Gossiping with Short Unreliable Messages. *Discrete Applied Mathematics*, 53:15–24, 1994.
- [9] F. Cristian and S. Mishra. The Pinwheel Asynchronous Atomic Broadcast Protocols. In *Proc. of the 2nd International Symposium on Autonomous Decentralized Systems*, Phoenix, AZ, 1995. Also: Technical Report CSE93-331, Department of Computer Science & Engineering, University of California, San Diego.

- [10] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, April 1996.
- [11] R. Feldmann, J. Hromkovic, S. Madhavapeddy, B. Monien, and P. Mysliewitz. Optimal Algorithms for Dissemination of Information in Generalized Communication Networks. In *Proceedings of Parallel Architectures and Languages Europe*, pages 115–130. Springer, 1992. Lecture Notes in Computer Science, 605.
- [12] S. Floyd, van Jacobson, S. McCanne, C. Liu, and L. Zhang. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *Proc. ACM SIGCOMM'95*, August 1995.
- [13] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, Department of Computer Science, Cornell University, 1998.
- [14] K. Guo and I. Rhee. Message Stability Detection for Reliable Multicast. In *Proc. of IEEE INFOCOM'2000*, March 2000.
- [15] K. Guo, R. van Renesse, W. Vogels, and K. Birman. Hierarchical Message Stability Tracing Protocols. Technical report, Department of Computer Science, Cornell University, 1997.
- [16] M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.
- [17] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [18] H.P. Katseff. Incomplete Hypercubes. *IEEE Transactions on Computers*, 37(5):604–608, May 1998.
- [19] D.W. Krumme. Fast Gossiping for the Hypercube. *SIAM Journal on Computing*, 21(2):365–380, April 1992.
- [20] R. Ladin, B. Liskov, L. Shrira, and G. Ghemawat. Providing Availability using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

- [21] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Kaufmann, 1992.
- [22] J. Liebeherr and T.K. Beam. HyperCast: A Protocol for Maintaining Multicast Group Members in a Logical Hypercube Topology. In *Proceedings of 1st International Workshop on Networked Group Communication (NGC '99)*, pages 72–89. Springer, July 1999. Lecture Notes in Computer Science, 1736.
- [23] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Toolkit for Building Fault-Tolerant Distributed Application in Large Scale. Technical report, Department d’Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.
- [24] S. Manor. Scalable Multicast in a Logical Hypercube. Master’s thesis, Department of Computer Science, Technion – Israel Institute of Technology, August 1999.
- [25] S. McCanne and S. Floyd. NS (Network Simulator) Home Page. <http://www-nrg.ee.lbl.gov/ns>.
- [26] L. Moser, P. M. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.
- [27] OMG. CORBA/IIOP Specification 2.4.2. formal/2001-02-33.
- [28] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharya. Reliable Multicast Transport Protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997. Special issue on Network Support for Multipoint Communication.
- [29] C. Plaxton, R. Rajaram, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.
- [30] Y. Saad and M.H. Schultz. Topological Properties of Hypercube. *ACM Transactions on Computers*, 37(7):867–872, July 1988.

- [31] D.S. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *Proceedings of 6th Distributed Memory Concurrent Computers Conference*, pages 398–403, 1991.
- [32] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, August 2001.
- [33] A. S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1996. Third Edition.
- [34] R. van Renesse. Masking the Overhead of Protocol Layering. In *Proc. ACM SIGCOMM'96*, pages 96–104, August 1996.
- [35] R. van Renesse, K. Birman, and S. Maffei. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.
- [36] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report Technical Report UCB/CSD-01-1141, Computer Science Department, U.C. Berkeley, April 2001.