Scalable Stability Detection Using Logical Hypercube

Roy Friedman  Shiri Manor  Katherine Guo
Department of Computer Science  Technion - Israel Institute of Technology  Bell Laboratories
Haifa 32000  ISRAEL  Networking Software Research
{roy, smanor}@cs.technion.ac.il  kguo@bell-labs.com

May 30, 1999

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 by both an analytic 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 physical topology of the underlying (physical) network.

Contact Author: Roy Friedman, Department of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, ISRAEL. Phone: +972-4-8294264. Fax: +972-4-9221128. E-mail: roy@cs.technion.ac.il.
1 Introduction

Reliable multicast has been recognized as a key feature in many distributed systems, as it allows to reliably disseminate the same message to a large number of recipients. Consequently, reliable multicast is supported by many middlewares such as group communication (Isis [5], Horus [24], Transis [8], Ensemble [1], Relacs [4], Phoenix [17], and Totem [19], to name a few), protocols like RMT [20] and SRM [9], and in the near future standards like CORBA [2]. 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 became 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 [13, 16] 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 maximal 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 implementing the protocol.

In this paper, we explore the performance and scalability of our proposed protocol, and compare it to other known stability detection protocols, namely, a fully distributed protocol, a coordinator based protocols, and a tree-based protocol [10, 11]. The comparison is done both analytically and by simulations. Our results show that the logical hypercube based protocol compares favorably with the other protocols we 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 were carried over randomly generated network topologies, and no attempt was 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 appeared 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 the system and/or network evolve.

1.1 Related Work

Many group communication toolkits, e.g., ISIS [5], Horus [24], and Ensemble [12], 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 [10, 11]. These include, e.g., 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 [11] it is assumed that the physical topology of the network matches the logical structure of the protocol, while in both [10] 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 [19] and the work of Cristian and Mishra [7]. 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 [10].

Hypercubes were originally proposed as an efficient interconnect for massively parallel processors (MPP) [13, 16], although to the best of our knowledge, no commercial computer was ever build with a hypercube architecture. A great body of research has been done on solving parallel problems on hypercubes, and due to lack of space we cannot mention even a small part of it (but see [13, 16]). However, to the best of our knowledge, we are the first to suggest to use a logical hypercube structure in order to deal with scalability issues in distributed environments.

Paper Roadmap: The rest of this paper is organized as follows: Section 2 describes the three known stability detection protocols to which we compare our approach. Section 3 details the logical hypercube based protocol, and compares the four protocols analytically. We present the simulation results in Section 4, and conclude with a discussion in Section 5.

2 Stability Detection Protocols

As discussed in Section 1, stability detection protocols aim to detect when messages become stable, i.e., have been received by all of their 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.

\footnote{In fact, a logical ring is used for communication over a shared access medium in the IEEE 802.4 standard [22], 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.}
In order to present the stability detection protocols, we introduce the following notation: We denote by $\text{ArrayMin}$ the element-wise array minimum of $n$ given arrays. That is, given $n$ arrays of length $k$, $R_1, \ldots, R_n$, $S = \text{ArrayMin}(R_1, \ldots, R_n)$ implies that for each $j \in [0, k-1]$, $S[j] = \min(R_1[j], \ldots, R_n[j])$.

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. As will become evident shortly, we are interested in the latency of each round, and determining the best value for $\Delta$ is beyond the scope of this paper. (But see [10].) We are now ready to present the protocols.

### 2.1 Coord Protocol

$\text{Coord}$ 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 $\text{ArrayMin}$ 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

$\text{FullDist}$ is a fully distributed protocol. That is, each node $i$ keeps a stability matrix of size $nxn$ 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 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$^2$ 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.

### 2.3 S_Coord Protocol

Several tree-structured protocols were introduced in [11]. In this work we take $S_{\text{Coord}}$ as representative of tree structured protocols, since it appeared to perform best in [11]. In $S_{\text{Coord}}$, 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

---

$^2$There is no significance to the choice of the first node. The first node that starts that protocol can be chosen in any deterministic way, e.g., according to nodes 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 $i$

$S_i := [0,0,...0]$;

< At coordinator $j$ >
Step 1
1: multicast($\text{start}$);
Step 2
2: wait until receive($\text{ack}, R_i$) from all nodes;
3: $S_j := ArrayMin (R_1, R_2, ..., R_n)$;
Step 3
4: multicast($\text{info}, S_j$);
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($\text{start}$) from coordinator;
Step 2
9: send ($\text{ack}, R_i$) to coordinator;
Step 3
10: wait until receive($\text{info}, S$) 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

\( \text{min}(R) \) – is 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
\( \text{receive}_i \) – the number of \text{info} messages node \( i \) has received in the current round

Initialization of node \( i \)

\( \text{receive}_i := 0; \)
\( \forall k, j, \text{where } k \neq i \) \( M_i[k,j] := 0; \)

Step 1

\( \text{At node } 0 \)
1: \( \text{wait} (\Delta); \)
2: \( \text{multicast} (\text{info}, \text{row}(M_0, 0)); \)

\( \text{At every node } k \neq 0 \)
3: \( \text{upon receive}(\text{info}, R) \text{ from node } 0 \) do
4: \( \text{row}(M_k, 0) := R; \)
5: \( \text{receive}_k := \text{receive}_k + 1; \)
6: \( \text{multicast} (\text{info}, \text{row}(M_k, k)); \)
7: \( \text{done}; \)

Step 2

\( \text{At every node } k \)
8: \( \text{upon receive}(\text{info}, R) \text{ from node } i \) do
9: \( \text{row}(M_k, i) := R; \)
10: \( \text{receive}_k := \text{receive}_k + 1; \)
11: \( \text{if } (\text{received}_k = n) \) then
12: \( \text{Label all messages received from every node } j \text{ with sequence number } P \leq \text{min} (\text{col}(M_k, j)) \text{ as stable}; \)
13: \( \text{receive}_k := 0; \)
14: \( \text{goto Step 1}; \)
15: \( \text{endif}; \)
16: \( \text{done}; \)

Figure 2: FullDist protocol
the minimum of the array it 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 $S_i$ 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 only. 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, $(X_{m-1}...X_1...X_0)$. Bit $X_k$ corresponds to dimension $k$. Two nodes $p$ with label $(X_{m-1}...X_1...X_0)$ and $q$ with label $(Y_{m-1}...Y_1...Y_0)$ are connected if and only if for some index $j$, $X_j \neq Y_j$ and $\forall i \in \{0, m-1\}, i \neq j, X_i = Y_j$. 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.
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 [13, 16, 21]. 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 [15]) 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 awaits 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 $j$. 

6
protocol for node \( i \)

**Notation**
- \( \text{children}_i \) — the indexes of node \( i \)'s children in the tree
- \( \text{rank}_i \) — number of children node \( i \) has in the tree
- \( \text{parent}_i \) — the id of node \( i \)'s parent in tree
- \( \text{ArrayMin} \) — 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
- \( \text{receive}_i \) — the number of \text{ack} messages node \( i \) has received in the current round

**Initialization**

\[
M_i := R_i, \quad S_i := [0,0,0,...0], \quad \text{receive}_i := 0;
\]

**Step 1**

\(< \text{At root}\>\)

1: multicast \((\text{start})\);

**Step 2**

\(< \text{At leaf } i>\)

2: upon receive \((\text{start})\) from root do
3: \quad send \((\text{ack}, R_i)\) to \( \text{parent}_i \);
4: \quad done;

**Step 3**

\(< \text{At internal node } i \text{ or root}>\)

5: upon receive \((\text{ack}, R)\) from \( j \) do
6: \quad \quad \quad \quad \quad M_i := \text{ArrayMin}(M_i, R);
7: \quad \quad \quad \quad \quad \text{receive}_i := \text{receive}_i + 1;
8: \quad \quad \quad \quad \quad \text{if} \ \text{receive}_i = \text{rank}_i \ \text{then}
9: \quad \quad \quad \quad \quad \quad \quad \text{send} \((\text{ack}, M_i)\) to \( \text{parent}_i \);
10: \quad \quad \quad \quad \quad \quad \quad \text{elseif} \ \text{< root> then}
11: \quad \quad \quad \quad \quad \quad \quad S_i := M_i;
12: \quad \quad \quad \quad \quad \quad \quad \text{multicast} \((\text{info}, S_i)\);
13: \quad \quad \quad \quad \quad \quad \quad \text{Label all messages received from every node } k \text{ with sequence number } P \leq S_i[k] \text{ as stable};
14: \quad \quad \quad \quad \quad \quad \quad \text{receive}_i := 0;
15: \quad \quad \quad \quad \quad \quad \quad \text{wait} (\Delta);
16: \quad \quad \quad \quad \quad \quad \quad \text{goto Step 1;}
17: \quad \quad \quad \quad \quad \quad \quad \text{endif};
18: \quad \quad \quad \quad \quad \quad \quad \text{endif};
19: \quad \quad \quad \quad \quad \quad \quad \text{endif};
20: \quad \quad \quad \quad \quad \quad \quad done;

**Step 4**

\(< \text{At every non-root node } i>\)

21: upon receipt of message \((\text{info}, S)\) do
22: \quad \quad \quad \quad \quad S_i := S; \quad \text{Label all messages received from every node } k \text{ with sequence number } P \leq S_i[k] \text{ as stable};
23: \quad \quad \quad \quad \quad \text{receive}_i := 0;
24: \quad \quad \quad \quad \quad \text{goto Step 1;}
25: \quad \quad \quad \quad \quad \text{done;}

Figure 3: \( S_{\text{Coord}} \) protocol
Figure 4: An example of a 4-dimensional hypercube ($H_4$).

- 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 that $i$ 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 $\text{ArrayMin}$ of $M_i$ and $M$. If $G_i$ contains all 1s, 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 $r_i$ they will have the same iteration number. Further messages from node $j$ to node $i$ will be delivered after $m_1$, and after $i$ 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.
protocol for node $i$

Notation

- $\text{neighbors}_i$ - indexes of node $i$'s neighbors in the hypercube
- $\text{rank}_i$ - number of neighbors node $i$ has in the hypercube
- $\text{ArrayMin}$ - element-wise minimum of the input arrays
- $\text{ArrayMax}$ - 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 numbers in this round
- $r_i$ - number of the current iteration
- $\text{receive}_\text{from}_i$ - the number of stability messages node $i$ has received in the current iteration

Initialization

- $M_i := R_i$
- $S_i := [0,0,0,...0]$
- $G_i[j] := 1 \ \forall \ j \neq i \ G_i[j] := 0$
- $r_i := 0; \text{receive}_\text{from}_i := 0$

< At every node $i$ >

Step 1

1: multicast(stability, $G_i, M_i, r_i$) to $\text{neighbors}_i$

Step 2

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

Figure 5: CubeFullDist protocol
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 [14], 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:

**Performance:** Minimize system diameter. System diameter determines the number of iterations in the stability detection protocol. Minimizing this factor imply faster stability detection.

**Fault tolerance:** Maximize the number of parallel shortest paths between two nodes. This factor determines the fault tolerance of the protocol. It has direct effect on performance when faults arise.

**Scalability:** Restrict the number of logical connections for each node to the limit of \(\log(n)\). The purpose of this restriction is to reduce protocol redundancy and traffic overhead.

We denote an incomplete hypercube by \(I_n^m\), 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 of multiple complete ones. There is a connection between node \(p\) in \(H_i\) to 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 \(H_3\) comprising of nodes 0xxx, \(H_2\) comprising of nodes 10xx, and \(H_1\) comprising of nodes 110x.

Our aim is to compensate for missing links in an incomplete hypercube \(I_n^m\) w.r.t. 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 \(I_n^m\) whose Hamming distance is 2 and are connected in \(H_m\) through a node that is missing in \(I_n^m\). 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, for each node \(z\) that belongs to \(H_m \setminus I_n^m\), \(G_z\) is defined as \(G_z = \{w|w \in I_n^m\ and H(z, w) = 1\}\). If \(G_z\) is empty or has only one member no connection is added. \(SG_z\) is now obtained by lexicography 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 w.r.t. \(H_3\). 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 $H_m \setminus P_n^m$. 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 $P_n^m$ is $m$, and by adding nodes we do not increase the diameter of the system. Hence, the described incomplete hypercube matches our goals. (A more detailed proof appears in the long version of this paper.)

### 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.

### 4 Experimental Performance

#### 4.1 Simulation Model

We used the ns [18] simulator to explore the behavior of the stability detection protocols described in Sections 2 and 3. We 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 between 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/Δ). Thus, the buffer size of unstable messages is proportional to the time it takes to detect message stability and the frequency 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 each message passes. Each message is counted once at each hop that it passes in 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, i.e., the average number of messages on all links in the network at any given time, and the maximum queue size, i.e., 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 total number of messages 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 Cube FullDist protocol. Note that both $S_{Coord}$ and Coord are not fault tolerant, so there is no point in looking at the effects of failures on them. Also, the performance of FullDist, as reported below, was so poor, that we decided to only show the results of failures on Cube FullDist.

The stability detection protocols were tested with several random generated network topologies. We used the random generator network GT-ITM [6] to build random network topologies with edge probability varying from 0.01 to 0.04; each protocol was run 6 times and in each run a different randomly generated topology was used. The results presented are the average of the 6 runs. The links in the simulations were chosen to be duplex 100Mbps in each direction with uniformly distributed delay between 0 to 1 ms. We assumed that the number of senders is 50, and the total number of nodes varied between 10 to 1900. The sequence number size was assumed to be 4 bytes, and thus the size of the array of stability information carried by $\text{stability, ack, and info}$ messages is 200 bytes. Also, the message header size was set to 32 bytes, large enough for most transport protocols [3, 23].

The tree degree in $S_{Coord}$ was set to $\log(n)$, i.e., each node has $\log(n)$ children. Node 0 was chosen as the coordinator of Coord, and as the root of the tree in $S_{Coord}$. In Cube FullDist, the logical hypercube was built according to node id’s binary representation. Since the network was generated randomly, the fitness of logical structure to the network is random as well. In Cube FullDist, when the number of nodes is not a power of two, the construction of incomplete hypercubes described in Section 3.3 is used.
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 w.r.t. complete hypercubes, since the graphs do not have any major jitters near and at system sizes that are power of two. (Note data point 100 vs. 128, data point 200 vs. 256 vs. 300, data point 500 vs. 512, and data point 1,000 vs. 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 10, 9, 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. 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, i.e., it takes time for the information to propagate in the system.

4.2.2 Network Load

Figure 8: RTT as a function of system size.
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.)
The average network load and the maximum queue size measured are reported in Figures 9 and 10, respectively. 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.

### 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. 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 underlining 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 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.

4.2.5 The Effect of Faults on CubeFullDist Performance

The effects nodes faults have on CubeFullDist performance can be seen in Figures 13 and 14. We tested the protocol with 0-5 node failures; faulty nodes were 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

In this paper we have described a logical hypercube based algorithm for stability detection, and compared it, both analytically and using simulations, to three other known stability detection protocols. This algorithm appears to be very scalable, and can tolerate node and link failures. The latency of detecting stability by this algorithm is slightly longer than using tree based protocols, but it is more robust.

Another interesting result of our work is that both the logical hypercube based protocol and the tree based protocol are insensitive to the underlying network topology. Recall that the network topologies we have chosen were randomly generated. It is likely that one can construct network
Figure 13: The effects of node failures on CubeFullDist performance.

Figure 14: The effects of node failures on CubeFullDist performance.
topologies that would fit one protocol better than the others. However, wide area networks tend to have general topologies. 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 [16], although looking at this problem in a more general context, such as the Internet, is an interesting research direction.

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. It would be interesting to look at adapting other aspects of reliable multicast, such as failure detection, causal delivery, and total ordering, to the use of logical hypercubes.

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, e.g., piggybacking stability messages on protocol messages, might be possible by tailoring the stability detection protocol to the multicast protocol.

References


Note that for the tree based protocol, our analysis and simulations assumed a balance tree. Even if the underlying network uses a spanning tree for routing, it is not very likely that this tree will be balanced.


