CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:1658–1678
Published online 25 July 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3321

A distributed virtual hypercube algorithm for maintaining scalable and dynamic network overlays

L. C. E. Bona[1,*,†], E. P. Duarte Jr.[1] and K. V. O. Fonseca[2]

[1]*Department of Informatics, Federal University of Parana, PO Box 19018, Curitiba Brazil*
[2]*Federal University of Technology—Parana, CPGEI, Av. Sete de Setembro 3165, Curitiba Brazil*

SUMMARY

Network overlays support the execution of distributed applications, hiding lower level protocols and the physical topology. This work presents DiVHA: a distributed virtual hypercube algorithm that allows the construction and maintenance of a self-healing overlay network based on a virtual hypercube. DiVHA keeps logarithmic properties even when the number of nodes is not a power of two, presenting a scalable alternative to connect distributed resources. DiVHA assumes a dynamic fault situation, in which nodes fail and recover continuously, leaving and joining the system. The algorithm is formally specified, and the latency for detecting changes and the subsequent reconstruction of the topology is proved to be bounded. An actual overlay network based on DiVHA called HyperBone was implemented and deployed in the PlanetLab. HyperBone offers services such as monitoring and routing, allowing the execution Grid applications across the Internet. HyperBone also includes a procedure for detecting groups of stable nodes, which allowed the execution of parallel applications on a virtual hypercube built on top of PlanetLab. Copyright © 2014 John Wiley & Sons, Ltd.

Received 24 January 2013; Revised 06 February 2014; Accepted 31 May 2014

KEY WORDS: virtual topology maintenance; distributed testing; network overlays; PlanetLab monitoring

1. INTRODUCTION

Overlay networks provide an infrastructure on which large-scale dynamic distributed systems and applications can be executed. They have been used to solve scalability problems [1, 2], support the execution of parallel applications [3], and offer monitoring services [4], as well as fault tolerance mechanisms to improve the robustness by allowing the system to reconfigure upon the detection of faults [5]. Several types of network overlays for different applications have been proposed, including overlays for supporting scalable P2P systems [6, 7], and for providing highly available routing services [8, 9]. These systems usually present an interface that allows connected resources, such as processors or data, to be shared while hiding lower level protocols and the physical topology to applications and users.

Overlay networks are usually based on structured virtual topologies, which are defined on top of the physical topology [10, 11]. When the underlying system or the upper level application changes, the network overlay topology needs to be adapted accordingly. These systems are thus dynamic, in the sense that the set of shared resources and processes may continuously change. In this work, we present the distributed virtual hypercube algorithm (DiVHA), which allows nodes to self-organize on a virtual topology that maintains the logarithmic hypercube properties under a dynamic situation,

*Correspondence to: L. C. E. Bona, Department of Informatics, Federal University of Parana, PO Box 19018, Curitiba Brazil.
†E-mail: bona@inf.ufpr.br

that is, as nodes leave and enter the system. The literature on the properties of other virtual topologies under a dynamic situation is scarce, an exception is Reference [12].

The hypercube is a scalable topology by definition, presenting important topological properties such as symmetry and logarithmic diameter, among others that are advantageous for deploying fault-tolerance [13–15]. It should be noted that characteristics that make it difficult to implement hypercubes in hardware [16, 17] do not hinder the deployment of a virtual hypercube: the number of nodes does not need to be a power of 2 and the cost of adding links, that are virtual, is very low. Even when the number of nodes is not power of 2, or when some nodes are not available, DiVHA is still able to maintain the hypercube logarithmic properties.

Each node running DiVHA keeps a local view of the topology, which is consistent given a proven time bound. Nodes are connected through virtual communication links, which are employed both to monitor events affecting the topology and to determine how information flows in the system. The monitoring process is organized in rounds; in each round, each node executes tests on its neighbors, and they exchange information about the virtual topology. Note that the testing interval is configurable, for example, it can be set to 30 s or 10 ms, depending on the application requirements. After an event occurs, it takes at most $log_2 N$ rounds, requiring at most $N * (log_2 N)^2$ tests for every working node to learn about the event and update its local topology view. The topology is adapted to the set of detected events in order to preserve two hypercube properties: (i) the logarithmic diameter, that is, the distance between every pair of node i and j is never greater than $log_2 N$ and (ii) the number of virtual communication links is never greater than $N log_2 N$.

A HyperBone [18] overlay network is constructed with DiVHA, which interconnects Internet hosts with virtual links that are edges of a virtual hypercube, implemented as persistent transmission control protocol (TCP) connections. HyperBone nodes are capable of executing processing tasks. In order to determine a set of nodes that present a behavior that is stable enough for the execution of processing tasks, HyperBone employs monitoring information obtained from DiVHA. HyperBone is capable of routing messages across the virtual hypercube, providing a scalable communication service that allows direct communication between any pair of nodes using routes with at most logarithmic length. All traffic between nodes is tunneled through the virtual links, making it easier to setup nodes behind restrictive firewalls.

HyperBone was implemented and executed on PlanetLab [19], the worldwide testbed that allows the deployment of network protocols and distributed systems at a global scale under real conditions. Experimental results show the system feasibility and report the execution of distributed parallel applications on a virtual hypercube spread across the world. Two other sets of experimental results are presented. The first set reports simulation results of DiVHA under several particular fault situations. The second set describes the application of HyperBone as a transport layer for Message Passing Interface (MPI) applications.

The rest of this paper is organized as follows. Section 2 presents the system model and the specification of the algorithm for building and maintaining the virtual hypercube, including example executions and formal proofs of correctness. Experimental results, including those obtained from the PlanetLab implementation, are presented in Section 3. Section 4 points to related work. Section 5 concludes the paper.

2. CONSTRUCTION AND MAINTENANCE OF A VIRTUAL HYPERCUBE

Overlay networks allow nodes of WANs to organize themselves in a way that makes communication more efficient or reliable. They provide specific services that allow for example searching or routing. Defining an approach for nodes of a large-scale network to self-organize is a difficult task, especially considering that this is a dynamic environment prone to failures and delays. In this section, we present the DiVHA based on which the HyperBone network overlay is constructed. HyperBone employs an active monitoring strategy, in which nodes execute tests in a distributed fashion so that only nodes that pass the tests get to participate in the overlay. We prove that even if nodes fail and are repaired and re-enter the system, the topology keeps its logarithmic properties.

DOI: 10.1002/cpe

This section starts with a description of the system model, and then the algorithms for keeping the network overlay are described and specified. Formal proofs of correctness follow, as well as example executions.

2.1. System model

Consider a system S consisting of a set of N nodes, $n_0,\ n_1, \ldots, n_{N-1}$. We alternatively refer to node n_i as *node* i. The system is assumed to be fully connected, that is, any pair of nodes can directly communicate with each other. We assume a partially synchronous system model, in which bounds on the transmission delay of messages and the relative speeds of processes exist but are unknown [20, 21].

Nodes execute tests in order to construct a *perceived* system state, based on which the set of nodes that form the overlay is determined. A test is implemented as a task assigned by tester to the tested node. The specification of the testing procedure often depends on the particular infrastructure technology on which the system is deployed. This means that the system deals with a failure class best characterized as incorrect computation failure [22], which occurs when a processor fails to produce the correct result in response to a correct input. Furthermore, as the system is partially synchronous and bounds on message delay are unknown, a test reply may take longer than expected to arrive. In order to allow nodes to determine a perceived state for a slow node, a test employs a *timeout* in order to define the maximum time interval employed to wait for a task output. The occurrence of a timeout is equivalent to receiving an incorrect output, in the sense that the tested node is considered to have failed.

Each node n_i is assumed to be in one of two states, according to the results of the tests to which it was subject: *working* or *failed*. An *event* is defined as a change in the state of a node, either from *failed* to *working* or from *working* to *failed*. The collection of states of all nodes is the system's perceived fault situation. A dynamic fault situation is considered, that is, the nodes can continuously alter between the *failed* and *working* states.

Nodes execute tests periodically at a configurable testing interval. This interval can range, for example, from a few milliseconds to several minutes. A node relies on its local clock. A *testing round* is defined as the period in which all working nodes have executed their assigned tests. The algorithm's latency is defined as the time required by all working nodes to receive information about a given event.

Our model allows nodes to experience instability periods which feature very quick changes in the perceived states [23]. We define *availability parameters* in order to allow the system to determine which nodes are presenting a 'stable' behavior. A *working* node is considered to be *unavailable* only if the state of that node toggles to *failed* and the node remains in the new state for a predefined length of time. Similarly, a *failed* node must toggle to and remain in the *working* state for a large enough time interval in order to be considered *available*. This strategy allows the identification of parts of the system, which present a more predictable behavior, also allowing the isolation of nodes that present a highly unstable behavior.

2.2. Building and maintaining the virtual hypercube

Let the directed graph $G = (S, T)$ be a *testing graph*. There is a directed edge $(i, j) \in T$, from node i to node j if node i is supposed to test node j in the next testing round. Thus, T represents the set of all tests to be executed in the system. If node i is faulty, then $\nexists(i, j)\forall j$. Let the $d_i(j)$ be the shortest distance from node i to node j in G. Let $h_{i,j}$ be the distance of node i to node j in the hypercube, that is, the number of bits in which their identifiers differ. $h_{i,j}$ is called the hypercube distance.

Nodes are organized in clusters for the purpose of testing. *Clusters* are lists of nodes, whose size increases progressively in powers of 2. A cluster with p nodes $n_j, \ldots, n_{j+p-1}$, where

p is a power of 2 and greater than 1, is formed up by the union of two clusters, one of them containing nodes $n_j, \ldots, n_{j+p/2-1}$ and the other cluster containing nodes $n_{j+p/2}, \ldots, n_{j+p-1}$. Figure 1 depicts a system with eight nodes organized in clusters. A cluster of 2^s nodes to be tested by node i, $C_{i,s}$ is defined recursively by the following expression:

DOI: 10.1002/cpe

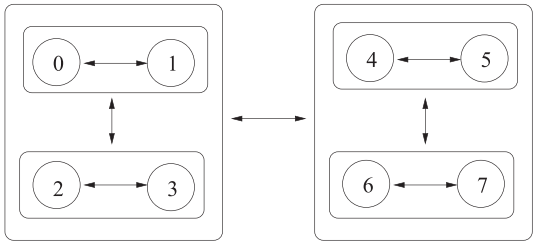

Figure 1. System with eight nodes grouped in clusters by DiVHA.

Distributed Virtual Hypercube Algorithm
$T \leftarrow \emptyset$
$T \leftarrow T \cup \{(i,j) \mid n_i$ is working and $h_{i,j} = 1\}$
for $s = 2$ to $log_2 N$ **do**
  for $distance = 2$ to s **do**
    for $i = 0$ to $N-1 \mid n_i$ is working **do**
      if $j \in C_{i,s}$ and $d_i(j) > s$ and $h_{i,j} = distance$ **then**
        add (i,j) to T
      end if
    end for
  end for
end for

Figure 2. DiVHA specification.

$$C_{i,s} = C_{j,s-1} \cup C_{i,s-1},\ j = i \oplus 2^{s-1} \text{ and } C_{i,1} = j,\ j = i \oplus 1$$

In order to determine the set of tests to execute, each node runs DiVHA, which is shown in Figure 2. Each node employs its local state information to determine which other nodes are working.

The DiVHA initially adds hypercube edges to the testing graph, corresponding to tests executed by all working nodes. If there are failed nodes, DiVHA employs a strategy of adding extra edges that preserves the logarithmic properties of the resulting topology, even though it is not a complete hypercube. In this case, working nodes share among themselves the task of executing tests on those nodes that have failed. These tests are determined on the basis of the distances computed on G and are described in the succeeding test.

At each iteration, the algorithm computes edges from every working node i, with i going from 0 to $N-1$, to every node in $C_{i,s}$. The algorithm checks existing paths within the smallest cluster to the largest, that is, the cluster index s is incremented from 2 to $log_2 N$. In a given cluster, the algorithm adds edges to nodes with the smallest to the largest (i.e., $2 \ldots s$) hypercube distance.

An extra edge is included to a node of a larger cluster only if there are no paths to that node with size at most s. In other words: an edge from node i to node $j \in Ci, s$ is added to T when there is no path from node i to node j or the path has size $> s$. Thus, edges are added between node 0 and every node in cluster $C_{0,s}$, between node 1 and every node in $C_{1,s}$, and so on until edges are added from node $N-1$ to the nodes that belong to $C_{N-1,s}$. An execution of DiVHA completes when there is at least a path with size $\leqslant s$ from every working node i to every node in $Ci, s, s = 1 \ldots N-1$.

The example in Figure 3 shows the edges added by DiVHA in a system in which all nodes with distance 1 from both node 0 and node 15 have failed. Gray nodes are faulty, and edges represent tests. Thin edges are hypercube edges, and thick edges are added by DiVHA. Note that in this case, DiVHA does *not* include edges to every node to which it has hypercube distance 2. After the first few edges are included, paths within the required limit appear to every other node. For instance, an extra edge is added from node 0 to node 5; but then, it is not necessary to include an extra edge to node 6, for the distance from node 0 to node 6 is now 2, passing through node 5.

DOI: 10.1002/cpe

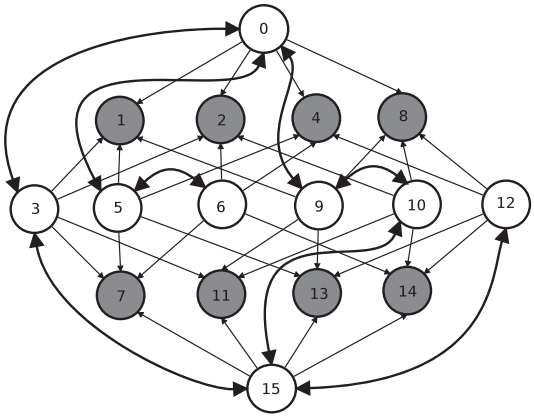

Figure 3. Edges added by DiVHA on an example system with several failed nodes.

HyperBone run at n_i
update T with DiVHA
loop
  for all $j \mid (i,j) \in T$ **do**
    test j
    if the state of node j has changed **then**
      $timestamp_i[j]++$
    end if
    if node j is working **then**
      for all $k \in d_j(k) < d_i(k)$ **do**
        if $timestamp_i[k] < timestamp_j[k]$ **then**
          $timestamp_i[k] = timestamp_j[k]$
          $d_i(k) = d_j(k) + 1$
        end if
      end for
    end if
    if new events have been detected **then**
      update T with DiVHA
    end if
  end for
  sleep until the next testing interval
end loop

Figure 4. Virtual hypercube construction and maintenance algorithm.

The algorithm shown in Figure 4 is run by every working node to determine the state of every other system node. At each testing interval, a working node i executes its tests as assigned by DiVHA as described in the preceding text. Thus, $\forall j|(i,j) \in T$: node i tests node j. When node i tests node j as *working*, the tester obtains information about every other node k such that $d_j(k) < d_i(k)$. Note that this guarantees both that $d_i(k) < log_2 N$ and that there are no cycles in G. In order to guarantee that node i keeps only the most recent information about the state of node k, timestamps are employed, as explained in the succeeding text.

A timestamp is implemented as a counter of perceived state changes at a given node, that is, it is incremented whenever a test is executed and the tester finds out that the tested node has changed its state. It is assumed that a node stays in a given state long enough so that it is tested at least once in that state. The timestamp is initially 0 and denotes a working state; thus, when the node is detected to have failed, the timestamp is incremented to 1, and so on. An even timestamp corresponds to a working node. An odd timestamp corresponds to a failed node. When node i tests a working node j, it gets the timestamps node j keeps about every other node $k : d(j,k) < log_2 N$. Then,

DOI: 10.1002/cpe

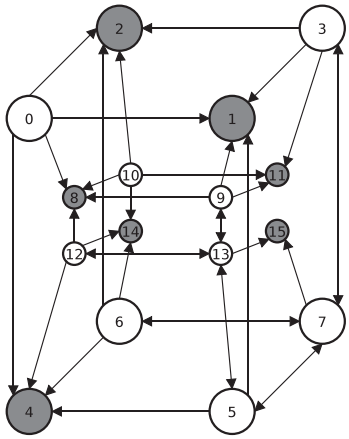

Figure 5. DiVHA initialization: hypercube edges are added.

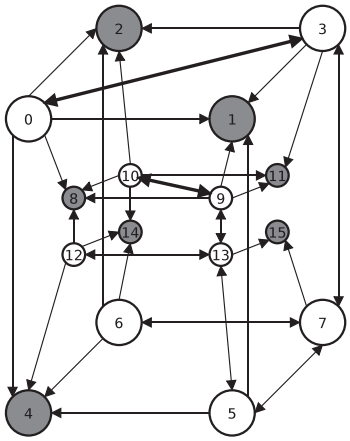

Figure 6. First iteration: paths sizes $\leqslant 2$ from n_i to every node in $C_{i,2}$.

node i updates its local information: $\forall k | timestamp_i[k] < timestamp_j[k] : timestamp_i[k] = timestamp_j[k]$. It is easy to devise an efficient implementation for transferring this information, for instance, node i can obtain only information that has changed since the last test.

2.3. Execution examples

In this subsection, we show examples of DiVHA executing on a system with 16 nodes ($logN = 4$). In the figures, a gray node is a failed node. The edges represent tests. Thin edges are hypercube edges, and thick edges are edges added by DiVHA, called extra edges. Figure 5 shows the first phase of the algorithm in which hypercube edges are added by working nodes.

Figure 6 shows that in the first loop iteration, the algorithm checks whether there are paths with size $\leqslant 2$ from every working node i to all nodes in each corresponding cluster $C_{i,2}$. For instance, node 0 checks whether there are paths of size at most 2 to node 2 and node 3. In the example, there is no path from node 0 to node 3, thus an extra edge is added: $0 \rightarrow 3$. In the same fashion, the following extra edges are added: $3 \rightarrow 0$, $10 \rightarrow 9$, $9 \rightarrow 10$.

The second loop iteration shown in Figure 7 checks paths with size $\leqslant 3$ from every working node i to nodes in cluster $C_{i,3}$. For instance, node 0 checks if there are paths with size at most 3 to nodes 4, 5, 6, and 7. The path from node 0 to node 4 has size 2; to node 7, size 2; to nodes 5 and 6, the paths have size 3. So, no extra edges are needed from node 0 to any node in $C_{0,3}$. Actually, in the example in this iteration, no extra edges are added.

DOI: 10.1002/cpe

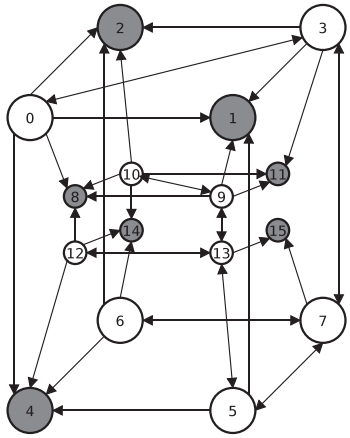

Figure 7. Second iteration: paths sizes $\leqslant 3$ from n_i to every node in $C_{i,3}$.

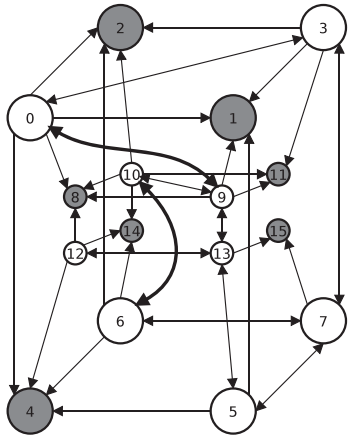

Figure 8. Third iteration: paths sizes $\leqslant 4$ from n_i to every node in $C_{i,4}$.

Finally, as shown in Figure 8, the last (third) iteration checks paths with size $\leqslant 4$ from every working node i to nodes in cluster $C_{i,4}$. For instance, node 0 checks if there are paths with size at most 4 to nodes 8, 9, 10, 11, 12, 13, 14, and 15. In the example the path from node 0 to node 9 has size at least 5, thus an extra edge is added. Considering all nodes, in this iteration the following other edges are also added: $9 \rightarrow 0$, $6 \rightarrow 10$ e $10 \rightarrow 6$, guaranteeing that paths have size at most equal to 4.

Given the topology in figure 8, figure 9 shows the information flow to node 0. Each group of encircled nodes represents a set of nodes from which node 0 is going to get information at once, in one test. A node i gets information from node j about all nodes k such that $d_j(k) < d_i(k)$. For example, node 0 tests node 3 and obtains information about node 5, node 6, node 7, node 11, and node 15. Node 0 tests node 9 and obtains information about node 9, node 10 and node 11, node 12, node 13, node 14, and node 5. Thus, node 0 obtains information about, for instance, node 15 from both node 3 and node 9. In order to guarantee that it updates the most recent information, timestamps must be checked.

Now we show how a different fault situation leads to a different topology and information flow. In Figure 10, every node is in the same state as in the aforementioned example, but node 4, which is working. Note that in this case, there is no need for the extra test that node 0 executed on node 9 in previous example in Figure 8, as there is now path $3 \rightarrow 4 \rightarrow 5 \rightarrow 13 \rightarrow 9$ of size 4 and node 9 is in $C_{0,4}$. On the other hand, node 0 needs to add an extra edge to node 10, as node 10 is in $C_{0,4}$, and without the extra edge, there was no path with size $\leqslant 4$.

DOI: 10.1002/cpe

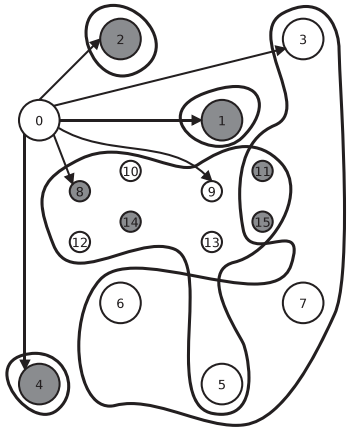

Figure 9. Information flow to node 0 in system of Figure 8.

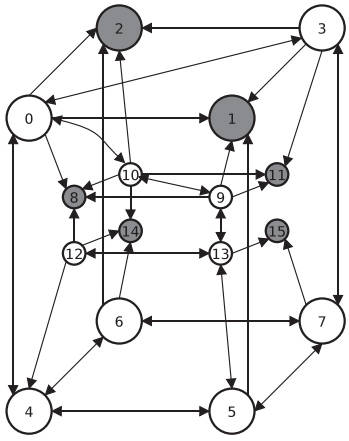

Figure 10. Same system but node 4 is working.

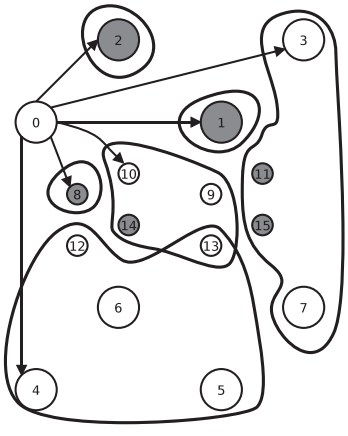

Figure 11. Information flow to node 0 in the system of Figure 10.

Figure 11 shows the information flow to node 0 considering the topology in Figure 10. It can be seen that the fact the node 4 is working causes a change in the information flow (compare Figures 9 and 11). For instance, in Figure 9, node 0 obtained information about node 9 from node 6. Now, there is a shorter path to node 6 through node 4.

DOI: 10.1002/cpe

2.4. Correctness proofs

In this section, we prove that DiVHA guarantees the system's logarithmic diameter even when the number of working nodes is not a power of 2 and that the total number of edges in $T(S)$ is at most $Nlog_2N$.

Lemma 1
In testing graph G computed by DiVHA, the distance from every working node to every other node is $\leqslant log_2N$.

Consider the algorithm depicted in Figure 2. The outmost loop adds edges guaranteeing that $\forall$ working node i and $j \in C_{i,s} : d_i(j) \leqslant s$. As $s \leqslant log_2N$, the lemma is proved.

Theorem 1
In testing graph $T(S)$ computed by DiVHA, the number of edges is at most $Nlog_2N$.

The proof is by induction on s, the cluster index; remember that $\forall i$ there are 2^s nodes in cluster $C_{i,s}$.

Basis: $\forall i$, there are at most two nodes in cluster $C_{i,1}$; when both nodes in the cluster are fault-free and test each other, the number of edges is maximum and is equal to 2.
Induction hypothesis: In cluster $C_{i,s}$, there are at most $s2^s$ edges.
Step: Cluster $C_{i,s+1}$ consists of the two clusters, $C_{i,s}$ and $C_{j,s}$, where $j = i \oplus 2^{s+1}$, plus the edges between the two clusters.

By the induction hypothesis, the number of edges in within these two clusters is $s * 2^{s+1}$.

Now, we count the number of edges between the clusters. Consider the algorithm in Figure 2. Without loss of generality, consider an edge from node $k \in C_{i,s}$ to node $l \in C_{j,s}$ as shown in Figure 12. From the proof of Lemma 1, the distance from any working node $\in C_{i,s}$ to node k is s. Thus, the distance from any working node in $C_{i,s}$ to l is $s + 1$. At this iteration, there is no other edge from another fault-free node in $C_{i,s}$ to l. Thus, there is at most one edge incident on every node of both clusters, the number of edges is $2 * 2^s = 2^{s+1}$.

Thus, the total number of edges in cluster $C_{i,s+1}$ is $s*2^{s+1}+2^{s+1}$, which is equal to $(s+1)*2^{s+1}$, completing the proof.

As $s \leqslant log_2N$, the maximum number of edges corresponding to the tests executed is $Nlog_2N$.

Theorem 2
A new event at any system node is detected by every working node in at most log_2N testing rounds.

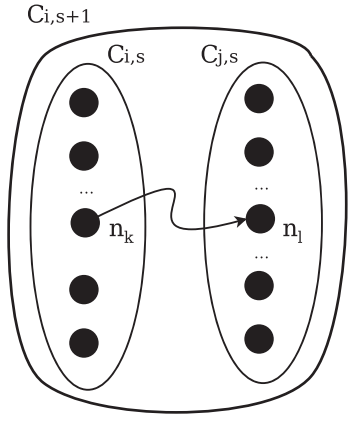

Figure 12. Extra edge added from node $k \in C_{i,s}$ to node $l \in C_{j,s}$.

DOI: 10.1002/cpe

Suppose an event occurs at node e. We show by induction that every working node at distance d from node e detects the event in at most d testing rounds.

Basis: every working node i such that $d_i(e) = 1$ detects the event at e in at most one testing round. This is trivial, for at each testing round, every working node tests all nodes within distance 1.

Induction hypothesis: every working i such that $d_i(e) = d$ detects the event at e in at most d testing rounds.

Step: every working node i such that $d_i(e) = d + 1$ detects the event at e in at most $d + 1$ testing rounds.

Consider j such that $d_j(e) = d + 1$; thus, there is a path $\{j =, i_1, i_2, \ldots, i_d, e\}$ with $d + 1$ edges from j to e. The path $\{i_1, \ldots, e\}$ has d edges; therefore, $d_{i_1}(e) = d$, and by the induction hypothesis, node i_1 detects the event at e in at most d testing rounds. As node j is adjacent to node i_1 and tests i_1 after at most one testing round and that $d(i_1, e) < log_2 N$ and $d\ (i_1, e) < d(j, e)$, j obtains information about any event at node e from i_1. Thus, j detects the new event at e in at most one more testing round after i_j, that is, in at most $d + 1$ testing rounds. As shown in Lemma 1 $\forall j, e : d_j(e) \leqslant log_2 N$; thus, in at most $log_2 N$ testing rounds, every node detects every event in the system.

3. EXPERIMENTAL RESULTS

In this section, experimental results are presented, which were obtained from three different implementations of DiVHA and HyperBone. The first set of experiments report simulation results of DiVHA under several particular fault situations. The second set describes the application of HyperBone as a transport layer for MPI applications. The third set describes the implementation and deployment of HyperBone in PlanetLab [19], the worldwide testbed that allows the deployment of network protocols and distributed systems at a global scale under real conditions. Experimental results show the system feasibility and report the execution of distributed parallel applications on a virtual hypercube spread across the world.

3.1. Simulation results

In this subsection, we present simulation results of a DiVHA under extreme situations: in the first experiment, millions of different fault situations were considered; in the second experiment, we checked the behavior of the system when from 1 to $N - 1$ nodes become unresponsive, and then one-by-one, they all recover.

The simulation was conducted using the discrete-event simulation language SiMulation Programming Language (SMPL) [24]. Nodes were modeled as SMPL facilities, and each node was identified by a SMPL token number. Two series of experiments were conducted. The first series of experiments shows the number of tests required as a function of the number of unresponsive nodes. The second series of experiments presented shows the algorithm's latency for a set of randomly generated fault situations.

3.1.1. Number of tests required. The purpose of these experiments is to measure the number of tests needed by a system running DiVHA. Hypercubes of six, seven, and eight dimensions were simulated, and about five million different fault situations were generated. For each fault situation, DiVHA was executed, and the system's testing graph built. The testing graphs determine the number of tests executed in the system, given that an edge corresponds to a test.

The graph, shown in Figure 13, presents the results obtained from simulations of a six-dimensional hypercube. The x-axis gives the number of unresponsive nodes in the fault situation. The y-axis presents the best-case and worst-case number of tests executed for the simulated fault situations. The best-case is the fault situation that required the smallest number of tests; the worst-case is the fault situation that required the largest number of tests. The results show that the number of tests decreases as the number of faulty nodes increases. Thus, we can conclude, as expected,

DOI: 10.1002/cpe

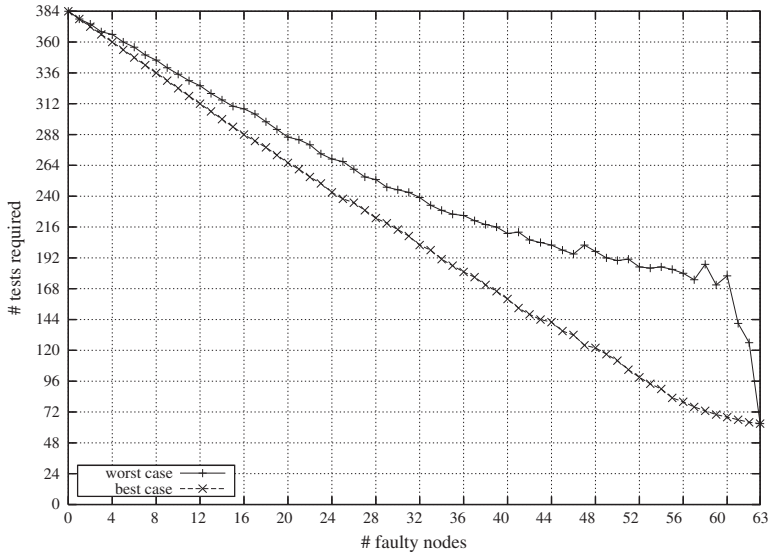

Figure 13. Number of tests required for a six-dimensional hypercube.

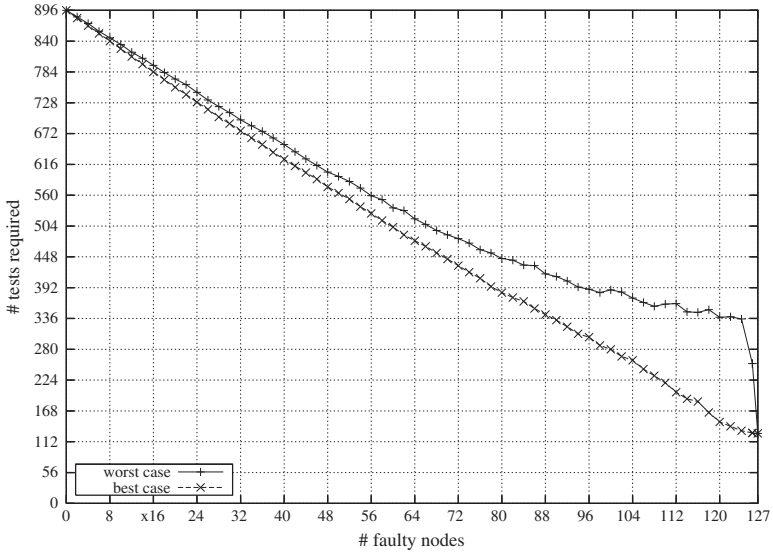

Figure 14. Number of tests required for a seven-dimensional hypercube.

that the worst-case for the number of tests corresponds to the fault situation in which all nodes are faulty-free.

In order to show the algorithm's scalability, the previous experiment was repeated for seven-dimensional and eight-dimensional hypercubes. The graph in Figure 14 shows the results for a seven-dimensional hypercube. The graph in Figure 15 shows the results for an eight-dimensional hypercube. Comparing the results obtained for the aforementioned systems, we can conclude that the algorithm presents a very similar behavior for different system sizes. The fact that the worst case number of tests is logarithmic, that is, at most $Nlog_2N$ tests are executed in a given testing round, was also confirmed by the simulation results.

3.1.2. Event detection latency. The latency is the number of rounds that all working nodes take to receive information about an event. The purpose of this series of experiments is to verify the latency to detect an event under different fault situations. The first experiment was conducted on a system

DOI: 10.1002/cpe

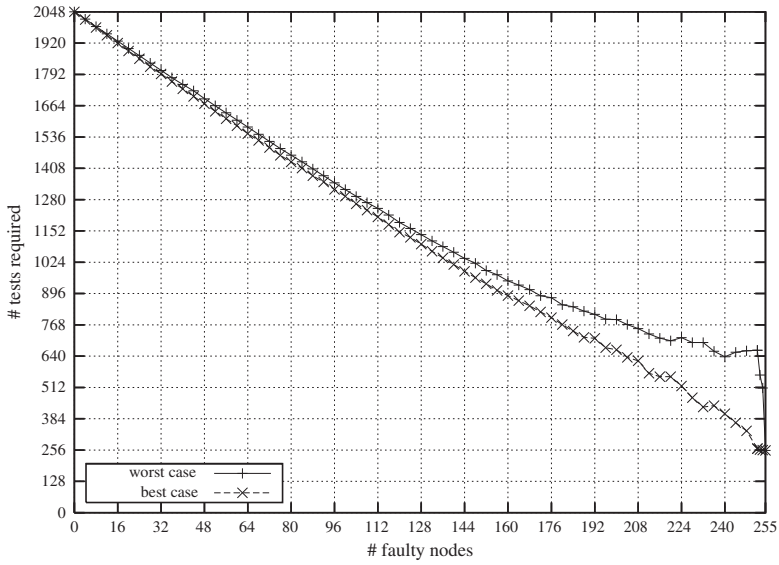

Figure 15. Number of tests required for systems with 256 nodes.

Table I. Latency for systems with 512 nodes.

Rounds	1	2	3	4	5	6	7	8
Frequency	148	518	1989	3628	2873	940	97	6

with 512 nodes (nine-dimensional hypercube). In this experiment, initially all nodes were working, then the number of faulty nodes was increased from 1 up to $N - 1$. When only one working node remained in the system, the other nodes were successively repaired until all nodes were working. The selection of the node that suffered a fault or repair event was random. This experiment was repeated 20 times measuring the latency of 10,200 events. The Table I shows the frequency distribution of the latency observed for the simulated events. Most of the events were detected within a number of rounds close to the mean expected latency, given the minimum and maximum possible values, that is, between 1 and $log_2 N$. If we sum up the number of events that were detected within 3, 4, and 5 rounds, we conclude that 8490 events were detected within those limits, corresponding to 83.4% of the total events simulated.

The same experiment was also conducted for systems of other dimensions. The graph in Figure 16 shows the latency observed for each system size. The x-axis gives the number of nodes in the system. The bars show the minimum and maximum theoretical latency for each system size. The line shows the most frequent latency registered. These results confirm that the latency growth is logarithmical and that most of the events are detected in about $(log_2 N)/2$ rounds.

3.2. An message passing interface transport layer based on HyperBone

This section introduces `ch_hyperbone`, a software device that allows users to run MPI-based parallel programs using HyperBone as the transport layer. The MPI standard [25] defines a set of functions and data types for parallel programming that hides transport details and data representation variation between the various nodes. MPI defines functions for sending and receiving data to/from other nodes and synchronization facilities.

The MPI over CHameleon (MPICH) library [26] is an open source implementation of the MPI standard. It provides support to a large range of transport devices, such as TCP/Internet protocol (IP), shared memory, and Globus [27], for instance. Its architecture consists of several layers. The lower layer is called abstract device interface (ADI) and defines a set of functions to be implemented

DOI: 10.1002/cpe

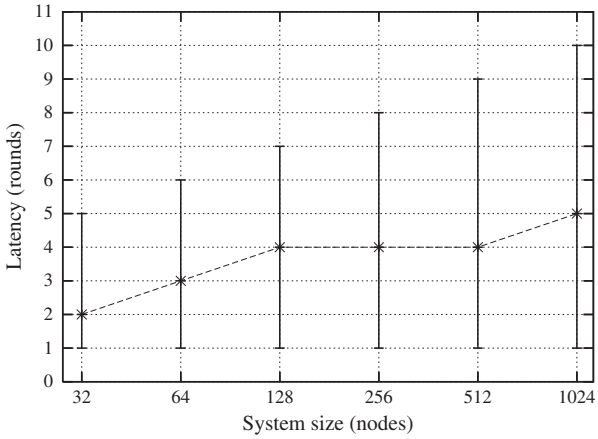

Figure 16. Latency for systems with several dimensions.

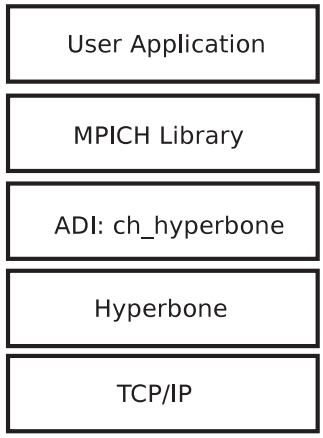

Figure 17. Execution layers.

by transport devices. Each transport device has its own ADI implementation, which may include the complete set of functions specified in the standard or just a subset of it. The upper layers use the available functions to perform the requested operations. For example, if the transport device does not implement multicast, the upper layers use a set of unicast calls, one for each peer supposed to be affected by the multicast operation. This architecture makes it easy to port the library to new transport devices.

The device we developed, `ch_hyperbone`, is the ADI implementation for using HyperBone as transport layer. It implements a small set of functions defined by the ADI specification, which consists of procedures for node initialization, unicast communication, and retrieval of the total number of nodes in the current execution.

The architecture is depicted in Figure 17. User applications employ high level functions provided by MPICH. These functions hide from user low level details, such as network address instead referring to other nodes only by their MPI node number. The MPICH library receives an application request and checks if the ADI implementation has built-in support for the corresponding operation. If so, it simply forwards the request to ADI. Otherwise, it combines the implemented functions in order to execute the requested operation. For instance, if the application issued a broadcast request, as `ch_hyperbone` does not implement native broadcast, it uses a set of unicast send operations, one for each node affected by the requested broadcast. `ch_hyperbone` receives each unicast send request, maps the MPI node number to the corresponding HyperBone node number, and for-

DOI: 10.1002/cpe

wards the message to the destination node using HyperBone. The message is forwarded through the hypercube using the existing TCP/IP connections until it reaches the target node.

The device was validated with two experiments. The first one measures the impact on the throughput caused by the overhead of routing the messages as the hypercube size increases. The second experiment uses MPI-PovRay [28], an image rendering application, to show the performance of a real application, running on HyperBone. The results are compared with the same application running with `ch_p4`, an ADI device that uses plain TCP/IP connections. Both experiments were executed several times, and the average figures obtained are presented.

In the first experiment, the throughput was evaluated measuring the time spent to transmit 100,000 messages of 8 KB each using the longest path in the hypercube. This is the worst case as in real applications, messages are expected to have shorter average paths. Figure 18 depicts the result. The first column shows the performance of the `ch_p4` device and achieved 653 Mbps. The second column shows that the throughput of a hypercube with just two nodes (1 hop) reached 629 Mbps. The three last columns display the results for four (2 hops), eight (3 hops), and 16 nodes (4 hops), which achieved 474, 463, and 328 Mbps, respectively.

The second experiment used MPI-PovRay to render a 3D image. This is not a network intensive application; it uses a master-slave architecture. The image used had 1000 × 500 pixels and was rendered using two, four, eight, and 16 nodes both with `ch_hyperbone` and `ch_p4`. The results are depicted in Figure 19 and show that `ch_hyperbone` results are similar to `ch_p4` results.

Using two nodes, `ch_hyperbone` spent 111 s and `ch_p4` 109 s. For four nodes, `ch_hyperbone` demanded 37 s and `ch_p4` 36 s. With eight nodes, `ch_hyperbone` used 17 s and `ch_p4` 20 s. Finally, using 16 nodes, `ch_hyperbone` needed 10 s and `ch_p4` 11 s.

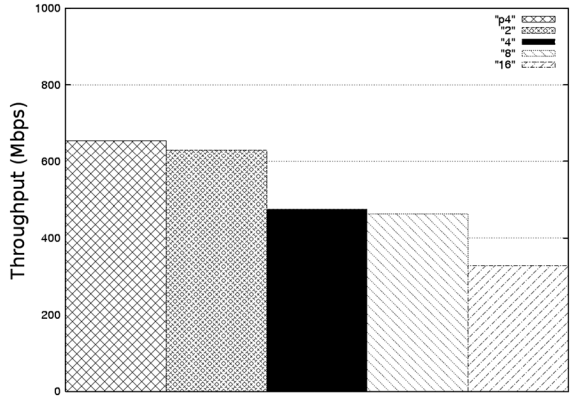

Figure 18. Impact on the throughput.

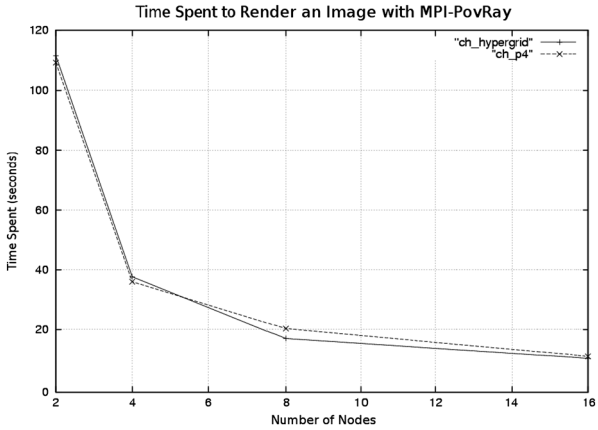

Figure 19. Time spent to render an image with MPI-PovRay.

DOI: 10.1002/cpe

The experimental results show that although the message routing through the hypercube introduces a significant overhead, which has an impact on the throughput achieved, HyperBone is suitable for applications that are not network intensive, such as MPI-PovRay.

3.3. The PlanetLab implementation

The DiVHA was implemented as a network overlay that was deployed on PlanetLab. HyperBone was implemented as a network daemon executing at each node of the system. Nodes communicate with each other through persistent TCP/IP connections corresponding to the virtual links defined by DiVHA. A test consists in sending a request message to the tested node, which is expected to reply within a predefined testing interval. If the reply does not arrive, the tested node is considered to be failed and may be either slow or have failed.

The overlay itself is transparent to the applications, that i, they communicate as if employing transport protocols directly. If an application wants to communicate using the virtual hypercube, it just needs to use an IP address reserved for HyperBone. For example, if the net-id address 10.0.0.0/8 is reserved for HyperBone, an application just needs to use an address that belongs to that range in order to communicate through the virtual hypercube. This feature is achieved using a `tun/tap` [29] interface that implements a virtual network device that receives packets from an user level application instead of receiving packets from datalink layer protocols.

The testbed chosen for obtaining experimental results with our HyperBone implementation was PlanetLab [19], which is a virtual open network on top of the Internet, consisting currently of nearly 700 hosts spread worldwide. The experiments described in the succeeding texts show how we evaluated the following aspects of HyperBone: the monitoring service, with a focus on defining a 'stable' state for nodes of a highly unstable environment, and the ability to execute parallel applications in the virtual hypercube.

3.3.1. Monitoring system. In order to evaluate the monitoring system, HyperBone was installed on 128 PlanetLab machines, and its execution was logged for more than 3 weeks. Different system parameters were tested. Considering the PlanetLab mean response times, the timeout for test reply messages was fixed in 7 s, and the testing interval in 10 s. Note that the testing interval is the maximum time a tester takes to discover that a tested node has failed. The results presented were obtained from monitoring the system execution with these parameters for 6 days.

One of the experiments aim at determining how long nodes stay in a given state before the occurrence of a new event. Given the relatively small timeout interval chosen, the nodes often did not

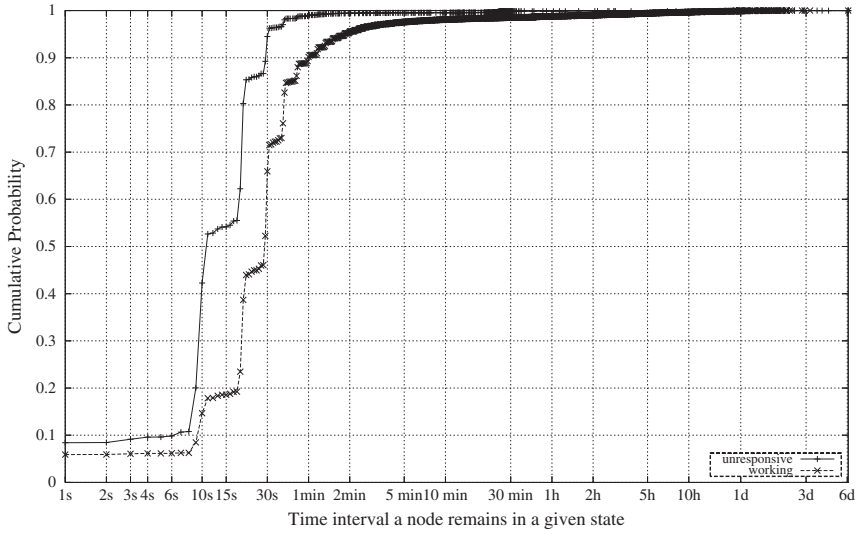

Figure 20. Probability distribution for the time interval a node remains working or failed.

DOI: 10.1002/cpe

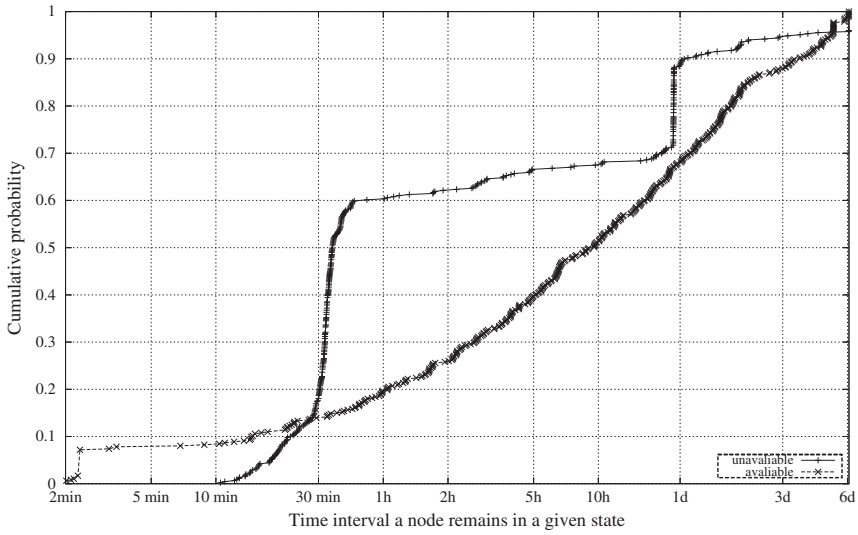

Figure 21. Probability distribution for the time a node remains available/unavailable.

reply within the required interval. As a result, a very large number of events was registered, nodes toggled state about 200,000 times in 6 days, resulting in an approximate rate of 1388 events/h. The graph in Figure 20 shows the cumulative probability distribution of the time expected for nodes to stay in a given state. In most cases, nodes stayed in the working state for less than 10 min before becoming failed and stayed failed for less than 120 s before toggling to the working state.

The very large number of events logged, showing that the environment is indeed highly unstable, led to the definition of parameters that allow HyperBone to determine which nodes are 'more stable', we say those nodes are *available*, whereas the others are *unavailable*. On the basis of system observation, as presented in Figure 20, we chose to consider a node to be *available* only after staying continuously for 600 s or more in the *working* state. A node is considered to be *unavailable* if it stays for more than 120 s in the *failed* state. These parameters have produced a significant decrease in the number of events HyperBone had to deal with. Just 1050 such events were registered, giving a rate of less than 8 events/h.

In practice, this way of classifying nodes allows the determination of a set of nodes that present a more stable behavior in comparison with other nodes. The granularity with which the system is seen increases. For example, an available node is going to be considered unavailable only if it stays more than 120 s failed; an unavailable node is going to be considered available only if it stays more than 600 s being considered working. In this way, available nodes that become silent for short periods are still considered available, and unavailable nodes that toggle to the working state for short intervals are still considered to be unavailable. The graph in Figure 21 shows the cumulative probability of nodes staying in one of these states. About 40% of the nodes stay for more than 1 h available before an event occurs, and about 80% of the nodes stay unavailable for more than 30 min before an event occurs.

Figure 22 shows the frequency distribution of the nodes availability. Despite the system instabilities, we observed that 47 nodes were available for between 80% and 90% of the monitoring time, and we even found 20 nodes that stayed in the available state for 90% of the time they were monitored. Other nodes were most of the time unavailable: they were slow, unreachable, or had crashed.

Finally, we measured the event propagation time required by HyperBone. We chose to monitor node 4, which at that period was quickly alternating its states: working/failed. Then, we checked how nodes at different distances from node 4 observed its state changes. These nodes were node 5, distance 1; node 7, distance 2; node 77, distance 3; node 118, distance 4; node 44, distance 5; node 122, distance 6; and node 123, distance 7. Figure 23 shows the results. For example, at approximately $x = 2h$, node 4 becomes unresponsive, and the figure shows the steps at which the other nodes detect this event. The clock used to date the time the events were detected was the local

DOI: 10.1002/cpe

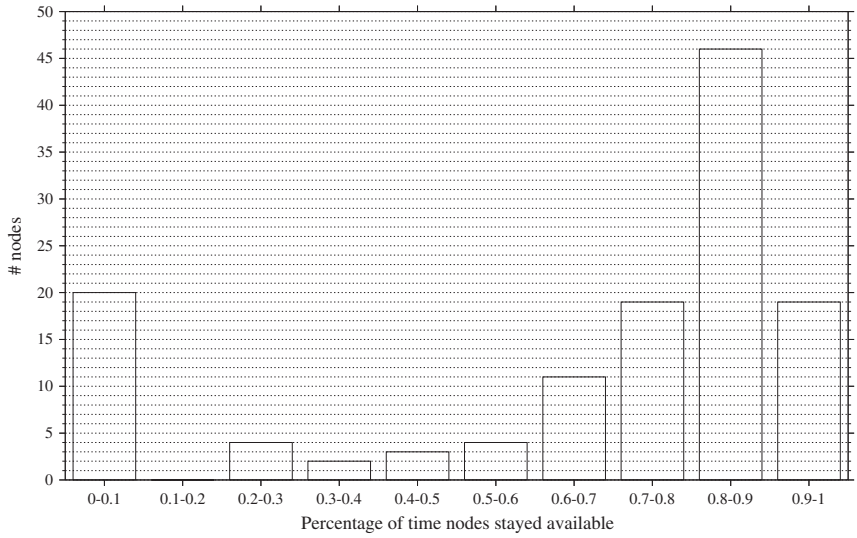

Figure 22. Length of time nodes stayed available/unavailable.

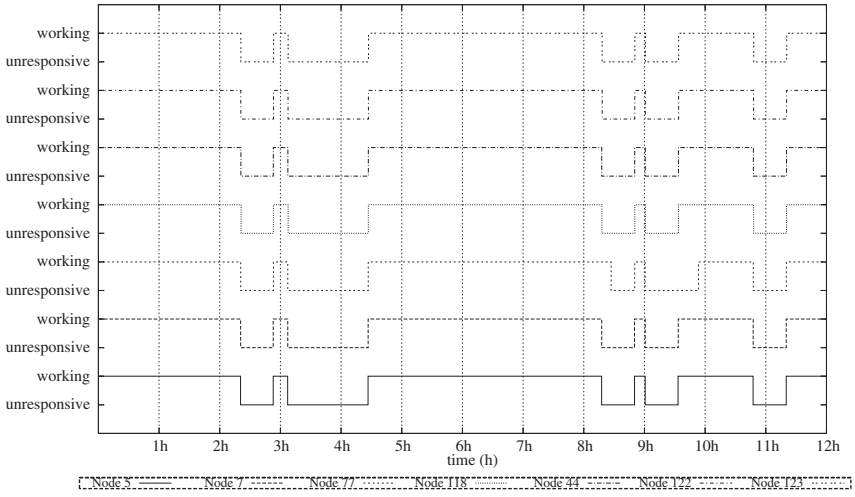

Figure 23. Time required by observers to detect the event at node 4.

clock synchronized with network time protocol. These results assert that events were discovered within the expected time interval.

3.3.2. Executing parallel applications. HyperBone's ability to execute applications was evaluated using a parallel application based on MPI run on the virtual hypercube. This type of application was chosen because HyperBone offers a platform for running scientific programs that can be solved efficiently with parallel algorithms and could take advantage of having access to a large number of processors. In order to run these experiments in PlanetLab, a virtual hypercube formed by 256 hosts was established. Those hosts were selected on the basis of the monitoring information previously described, and only *available* nodes were selected for processing tasks.

The purpose of the first experiment was to check that the MPI communication primitives written for HyperBone were functional. MPIbench [30] was employed for that purpose, a *benchmark* that exercises the main MPI functions, including point-to-point (`MPI_Send` and `MPI_Receive`) and group communication primitives, such as broadcast (`MPI_Broadcast`). Table II shows the results of the tests carried out, the rows show the tests (MPIbench benchmark), and the columns show the message size used in each test. Performance was not our goal here. Our objective was to execute suc-

DOI: 10.1002/cpe

Table II. MPIbench test results.

Message size	8 K	16 K	32 K	64 K
Latency in milliseconds	25.65	43.23	82.23	207.18
Round-trip in transaction/second	11.83	8.12	5.13	2.53
Bandwidth in KB/s	232.29	312.55	341.39	375.95
Broadcast rate in KB/s	54.81	47.13	34.26	36.26
All-to-all message rate in KB/s	100.40	100.80	121.13	98.11

Table III. Parallel password search results.

# Nodes	Passwords/second
16	31368.14
32	42968.33
64	50200.20

cessfully the existing applications of MPIbench without modifying a line on the virtual hypercube spanning the globe built on top of the very unstable environment that PlanetLab revealed to be.

The second experiment carried out had the goal of executing a useful MPI parallel application using HyperBone. That was an MPI application that parallelized the task of finding out a password in given space of search. Table III shows results (number of passwords checked per second) for systems with 16 and 32 nodes. We can observe that the system scales well as the number of nodes increases from 10 to 32 and then from 32 to 64.

4. RELATED WORK

Several overlay networks have been proposed on the basis of different topologies, such as mesh [11], ring [10], d-dimensional torus [31], and butterfly [32, 33]. Many of these overlay networks aim at offering services to P2P systems, for example, organizing the nodes for providing efficient content search and avoiding the use of flooding algorithms to propagate queries. Distributed hash tables are one of the most common techniques to deploy these search facilities [34]. Those systems are different from HyperBone because often the topology is a function of the stored objects and not the hosts themselves.

On the other hand, HyperBone offers monitoring, routing, and communication services that allow the execution of distributed applications and is capable of hiding the substrate network instability. Other systems are also able to deal with the system instability [32, 33], but, again, the main target of those systems is to preserve and make available the content stored in the network.

The fault detection issue in overlay networks is treated by [35], where different classes of fault detection algorithms are evaluated. The paper employs analytical models, simulations, and experimentation using metrics such as detection time, false positive probability, overhead and packet loss rates. The conclusions of that work evidence the importance of efficient monitoring strategies.

Unreliable failure detectors were proposed by Chandra and Toueg [36] as a distributed oracle that gives (possibly incorrect) hints about which processes of the system have crashed. In [20], ring-based algorithms are presented for the implementation of failure detectors in a partially synchronous system. This contribution is similar to ours, in the sense that a specific topology is proposed to organize and reduce the cost of monitoring.

In [9], resilient overlay networks (RON) are introduced, whose objective is to offer alternative routes for Internet-based applications. RON allows the application to survive through Internet routing instability periods. Application nodes employ monitoring information about virtual links in order to decide whether to send packets directly through Internet routes or to employ the alternative routes provided by RON. The system architecture is based on a full mesh. As pointed by the authors, RONs do not scale well for systems with more than 50 nodes.

DOI: 10.1002/cpe

Scalability is also a concern of [37], which presents an overlay network using the hypercube topology. The algorithm classifies hypercube positions as free or covered, executing join and departure requests. In order to maintain the system integrity, broadcast messages are sent informing all nodes when join, and departure requests are executed. The system is different from HyperBone because just a single event, join or departure of a node, can occur at a given time. The authors state that supporting multiple simultaneous events is future work.

In [38], Hi-ADSD, a diagnosis algorithm [39] based on a hypercube is presented. Like DiVHA, Hi-ADSD guarantees the logarithmic diameter, but the number of virtual edges can reach $O(N^2)$. DiVHA was first presented in [18] in the context of the HyperBone overlay network. In that paper, a preliminary specification of the algorithm was given, together with draft proofs of correctness. In the present work, the algorithm is presented in a definitive form, with complete proofs. That same early version of the algorithm was also deployed for network monitoring as described in the short documents [40, 41].

Some overlay networks have been proposed for distributed computing environments [27, 42], for example, to manage scalability problems. In [43], an overlay network is proposed to implement a resource discovery service, offering a scalable alternative to the grid's own search service. In [44], a scalable and fault tolerant communication topology called binomial graph is proposed. The paper discusses several fault tolerant routing algorithms for multicast and broadcast that can be efficiently used in HPC applications such as FT-MPI and OpenMPI.

In [45], a topology maintenance algorithm for one-hop DHTs is proposed. The authors proposed a new reliable multicasting algorithm called *aecast*, employed for dissemination of topology changes. *Aecast* combines best-effort multicasting and controlled flooding.

In [46], the overlay network construction problem is tackled, an gossip-based algorithm called T-Man is proposed. An initial overlay network with random links is the algorithm's starting point. Then, all nodes communicate to each other in a randomly based order to improve the quality of their neighbor set converging to the final topology.

5. CONCLUSION

Overlay networks are an important building block for running distributed applications. Keeping a large-scale structured overlay network under a dynamic fault situation is a nontrivial task. Although several overlay topologies have been presented in the past decade, few formal results have been presented on how these topologies adapt to a dynamic situation. In other words, it is unknown when the topology will present the desired properties after events occur.

In this work, we presented the DiVHA, which keeps the hypercube logarithmic properties even when the number of nodes is not a power of two, and under a dynamic fault situation, in which nodes fail and recover continuously, leaving and joining the system. The algorithm is formally specified, and the latency for detecting changes and the subsequent reconstruction of the topology is proved to be bounded. We proved that DiVHA guarantees that the topology diameter is equal to $log_2 N$ and the total number of edges is at most equal to $N log_2 N$. Furthermore, the algorithm still guarantees that the distance of any node to all other nodes is still $log_2 N$. Simulation results are presented showing how DiVHA performs under several particular fault situations.

A HyperBone overlay network constructed with DiVHA was also presented, which interconnects Internet hosts with virtual links that are edges of a virtual hypercube, implemented as persistent TCP connections. HyperBone is capable of monitoring nodes and routing messages across the virtual hypercube, providing a scalable communication service that allows direct communication between any pair of nodes using routes with at most logarithmic length. All traffic between nodes is tunneled through the virtual links, making it easier to setup nodes behind restrictive firewalls.

HyperBone was implemented and executed on PlanetLab [19], the worldwide testbed that allows the deployment of network protocols and distributed systems at a global scale under real conditions. Experimental results show the system feasibility and report the execution of distributed parallel applications on a virtual hypercube spread across the world. Another set of experiments describes the application of HyperBone as a transport layer for MPI applications.

DOI: 10.1002/cpe

We are currently working on a one-hop DHT based on DiVHA. As the algorithm allows nodes to have a view (given proven bounds) of the topology connecting working nodes, it encourages the definition of a hierarchical key space division based on the virtual hypercube. Working on a strategy to configure the availability settings dynamically, that is, to change in response to the system's conditions and also past behavior, is also planned. Future work also includes implementing group communication facilities on top of the virtual hypercube.

ACKNOWLEDGEMENTS

HyperBone was implemented as a transport layer for MPI by Samuel Lucas Vaz de Melo. This work was partially supported by grant 311221/2006-8 from the Brazilian Research Agency (CNPq).

REFERENCES

1. Cai M, Frank M, Chen J, Pedro S. MAAN: a multi-attribute addressable network for grid information services. *Journal of Grid Computing* 2004; **2**(1):3–14.
2. Huang L. VIRGO: virtual hierarchical overlay network for scalable grid computing. *Advances in Grid Computing - EGC 2005 Lecture Notes in Computer Science* 2005; **3470**:401–409.
3. Genaud S, Rattanapoka C. P2P-MPI: a peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing* 2007; **5**(1):27–42.
4. Lowekamp B, Miller N, Karrer R, Gross T, Steenkiste P. Design, implementation, and evaluation of the remos network monitoring system. *Journal of Grid Computingi* 2003; **1**(1):75–93.
5. Hwang S, Kesselman C. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing* 2003; **1**(3):251–272.
6. Clark D. Face-to-face with peer-to-peer networking. *IEEE Computer* 2001; **34**(1):18–21.
7. Doval D, O'Mahony D. Overlay networks: a scalable alternative for P2P. *IEEE Internet Computing* 2003; **7**(3):2–5.
8. Amir Y, Danilov C. Reliable communication in overlay networks. *IEEE International Conference on Dependable Systems and Networks (DSN 2003)*, San Francisco, CA, USA, 2003; 511–520.
9. Andersen DG, Balakrishnan H, Morris MFKR. Resilient overlay networks. *Operating Systems Review* 2001; **35**(5): 131–145.
10. Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H. Chord: a scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM*, San Diego, CA, USA, 2001; 149–160.
11. Rowstron A, Druschel P. Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science* 2001; **2218**:329–350.
12. Li X, Misra J, Plaxton CG. Concurrent maintenance of rings. *Distributed Computing* 2006; **19**(2):126–148.
13. Tien S, Raghavendra CS. Algorithms and bounds for shortest paths and diameter in faulty hypercubes. *IEEE Transactions on Parallel and Distributed Systems* 1993; **4**(6):713–718.
14. Krull JM, Wu J, Molina AM. Evaluation of a fault-tolerant distributed broadcast algorithm in hypercube multicomputers. *20th ACM Annual Computer Science Conference*, Kansas City, MO, USA, 1992; 459–462.
15. Tzeng NF, Chen HL. Structural and tree embedding aspects of incomplete hypercubes. *IEEE Transactions on Computers* 1994; **43**(12):1434–1439.
16. Saad Y, Schultz MH. Topological properties of hypercubes. *IEEE Transactions on Computers* 1988; **37**(7):867–872.
17. Hayes JP, Mudge TN, Stout QF, Colley S, Palmer J. Architecture of a hypercube supercomputer. *IEEE International Conference on Parallel Processing*, University Park, PA, USA, 1986; 653–660.
18. Bona LCE, Duarte EP, Jr., Fonseca KVO, Mello SLV. HyperBone: a scalable overlay network based on a virtual hypercube. *The 8th International Symposium on Cluster Computing and the Grid (CCGRID'2008)*, Lyon, France, 2008; 58–64.
19. PlanetLab: an Open Platform for planetary-scale services. (Available from: http://www.planet-lab.org) [Accessed on 7 July 2014].
20. Larrea M, Fernndez A, Arvalo S. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers* 2004; **53**(7):815–828.
21. Verissimo P. Travelling through wormholes: a new look at distributed system models. *ACM SIGACT News* 2006; **37**(1):66–81.
22. Laranjeira LA, Malek M, Jenevein RM. Nest: a nested-predicate scheme for fault tolerance. *IEEE Transactions on Computers* 1993; **42**(11):1303–1324.
23. Duarte EP, Jr., Garrett T, Bona LCE, Carmo RJS, Zuge A. Finding stable cliques of PlanetLab nodes. *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10 DCCS)*, Chicago, IL, USA, 2010; 317–322.
24. MacDougall MH. *Simulating Computer Systems: Techniques and Tools*. The MIT Press: Cambridge, MA, 1987.
25. Message Passing Interface Forum. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications* 1994; **8**(3/4):165–414.

DOI: 10.1002/cpe

26. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 1996; **22**:789–828.
27. Foster I, Kesselman C (eds.) *The Grid* (2nd edn). Morgan Kaufmann: San Francisco, CA, USA, 2003.
28. MPI-Povray—distributed Povray using MPI message passing. (Available from: http://www.verrall.demon.co.uk/mpipov/) [Accessed on 9 July 2008].
29. Universal TUN/TAP driver. (Available from: http://vtun.sourceforge.net/tun/) [Accessed on 7 July 2014].
30. MPIbench Home Page. (Available from: http://icl.cs.utk.edu/projects/ llcbench/mpbench.html) [Accessed on 7 July 2014].
31. Ratnasamy S, Francis P, Handley M, Karp R, Shenker S. A scalable content addressable network. *ACM SIGCOMM*, San Diego, CA, USA, 2001; 161–197.
32. Fiat A, Saia J. Censorship resistant peer-to-peer content addressable networks. *13th ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA, USA, 2002; 94–103.
33. Saia J, Fiat A, Gribble S, Karlin AR, Saroiu S. Dynamically fault-tolerant content addressable networks. *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, 2002; 270–279.
34. Balakrishnan H, Kaashoek MF, Karger D, Morris R, Stoica I. Looking up data in P2P systems. *Communications of the ACM* 2003; **46**(2):43–48.
35. Zhuang S, Geels D, Stoica I, Katz RH. On failure detection algorithms in overlay networks. *24th IEEE Infocom*, Miami, FL, USA, 2005; 13–17.
36. Chandra T, Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 1996; **43**(2):225–267.
37. Schlosser M, Sintek M, Decker S, Nejd W. HyperCuP - hypercubes, ontologies and P2P networks. *LNCS* 2002; **2530**:112–124.
38. Duarte EP, Jr., Nanya T. A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transaction on Electronic Computers* 1998; **47**(1):34–45.
39. Masson GM, Blough D, Sullivan GF. *System Diagnosis, Fault-Tolerant Computer System Design* Pradhan DK (ed.). Prentice-Hall: Upper Saddle River, NJ, USA, 1996.
40. Bona LCE, Duarte EP, Jr., Fonseca KVO. A scalable monitoring strategy for highly dynamic systems. *11th IEEE/IFIP Network Operations and Management Symposium (NOMS'08), Short Paper*, Salvador, Brazil, 2008; 919–922.
41. Bona LCE, Duarte EP, Jr., Fonseca KVO. HyperBone: a scalable overlay network based on a virtual hypercube. *11th IEEE/IFIP Network Operations and Management Symposium (NOMS'08), Dissertation Digest Paper*, Salvador, Brazil, 2008; 1025–1030.
42. Litzkow M, Livny M, Mutka M. Condor—a hunter of idle workstations. *8th International Conference of Distributed Computing Systems (ICDCS'88)*, San Jose, CA, USA, 1988; 104–111.
43. Hauswirth M, Schmidt R. An overlay network for resource discovery in grids. *16th International Workshop on Database and Expert System Applications (DEXA'05)*, Copenhagen, Denmark, 2005; 343–348.
44. Angskun T, Bosilca G, Dongarra J. Binomial graph: a scalable and fault-tolerant logical network topology. *The Fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA'07)*, Niagara Falls, Canada, 2007; 471–482.
45. Risson J, Harwood A, Moors T. Topology dissemination for reliable one-hop distributed hash tables. *IEEE Transactions on Parallel and Distributed Systems* 2009; **20**:680–694.
46. Jelasity M, Montresor A, Babaoglu O. T-man: gossip-based fast overlay topology construction. *Computer Networks* 2009; **53**:2321–2339.

DOI: 10.1002/cpe