HyperD: Analysis and Performance Evaluation of a Distributed Hypercube Database

Andi Toce  
The Graduate Center (CUNY),  
365 Fifth Ave.,  
New York, NY 10016, USA

Abbe Mowshowitz  
Akira Kawaguchi  
The City College of New York (CUNY),  
Convent Ave. at 138th St.,  
New York, NY 10031, USA

Paul Stone  
Patrick Dantressangle  
Graham Bent  
IBM, Emerging Technology Services,  
Hursley Park, MI137,  
Winchester, Hants, SO21 2JN, UK

Abstract—A Dynamic Distributed Federated Database (DDFD) offers advantages in a variety of applications. In previous work we introduced HyperD, a DDFD whose nodes are arranged in a virtual hypercube structure. We showed that this engineered structure reduces the cost of communication and facilitates processing of complex queries involving joins and semijoins. Moreover, we showed that the additional cost of maintaining this engineered structure is often outweighed by the savings produced. In this paper we present an improved protocol for building and maintaining HyperD. This protocol builds on the version of HyperD introduced earlier, and describes the implementation in greater detail. The main contributions of this paper are theory-based performance analysis and experimental results confirming the theoretical performance characteristics of the HyperD protocol. We show that building a hypercube DDFD is feasible and proves to be beneficial in terms of reducing the communication cost.

I. INTRODUCTION

The exponential increase in the volume of data that is generated every day has prompted researchers to look for new ways to structure database systems. Dynamic distributed systems are increasingly popular in database research. In such systems the data collected are stored in different locations using heterogeneous storage devices. To access these data and allow users to perform queries efficiently a variety of approaches have been adopted. Well understood topologies such as the hypercube have been found to be useful in both centralized and fully distributed systems. Distributed environments pose many challenges. In particular, when deploying and maintaining a fixed structure in a dynamic environment we need to resolve issues such as scalability, churn, and fault tolerance while efficiently exploiting the benefits of the topology.

Hypercubes are widely used in distributed systems due to a wealth of desirable properties. An n-dimensional hypercube is a n-regular graph of (relatively low) diameter n; it is highly symmetric, having a transitive automorphism group, and thus allows for balancing the network load; it can be expanded easily to a higher dimension and is fault tolerant. In addition, a number of efficient communication algorithms exist for hypercubes.

In this paper we describe an improved protocol for building and maintaining HyperD, a Dynamic Distributed Federated Database (DDFD) [2] structured as a virtual n-dimensional hypercube. To deal with missing hypercube positions we allow nodes to take temporary control of more than one hypercube position. By maintaining a virtual complete hypercube we are able to make efficient use of known communication algorithms which take advantage of the hypercube structure. We conduct experiments to test the performance of the HyperD protocol and to identify those conditions in which this approach offers savings in terms of bandwidth utilization.

II. BACKGROUND AND PREVIOUS WORK

A DDFD is a fully distributed relational database model for large networks of heterogeneous database systems. This architecture is well suited to a broad range of database applications where data is stored locally at each participating node while being accessible to all other nodes in the network. The research reported here has been conducted with reference to a particular DDFD, namely, the Gaian Database (GaianDB) [2]. The GaianDB grows by a process of preferential attachment, resulting in a scale free network. The advantages of using the GaianDB model in improving query processing in a distributed environment have been shown in [3]. To further improve the performance of the GaianDB we have been studying the use of fixed structures such as hypercubes as a model for the network. A network whose topology is known offers a number of advantages. In [13], [14] we demonstrated potential advantages of an engineered hypercube topology for efficient processing of queries in a distributed environment. In particular, we identified as a critical factor the internode distances in processing complex queries requiring multiple joins and semijoins. In order to verify any theoretical predictions it is necessary to come up with a practical implementation of a hypercube network. Examples in the literature are plentiful and several methods can be used to grow and maintain a hypercube. In [18] we presented the first version of HyperD, which adopts a growth model that is well suited to DDFDs. The steps shown in [18] serve as building blocks for the protocol described in this paper.

The choice of the hypercube as a model for our implementation is not arbitrary. Hypercubes have been extensively used in parallel processing. An example can be found in [11]. A
detailed survey of the use of hypercubes in parallel processing is found in [8]. Hypercubes have also found application in sensor networks [6] and wireless communication networks [5], [7]. With the advent of peer to peer (P2P) networks, a variety of structured graphs including hypercubes have been used as a models for connecting peers. A hypercube P2P network is proposed in [16] where a distributed algorithm arranges the nodes in an implicit hypercube. In [1] a hypercube-based P2P overlay for a Distributed Hash Table system is constructed allowing efficient lookup operations and providing a resilient structure against attacks and frequent network changes. A similar approach is also used in [12]. For a comprehensive study of the topological properties and formal definition of hypercubes see [10], [9], [15]. Our approach differs from most previous work in that the integrity of the hypercube is maintained by allowing nodes to take control of multiple hypercube positions. At any given time labels representing these hypercube positions are shared among nodes and redistributed when network changes occur. In addition, we employ a method of preferential attachment to assign labels to new arriving nodes so that virtual neighbors are also relatively close in the underlying physical network. Maintaining the integrity of the hypercube allows us to adopt without major modifications traditional hypercube communication algorithms.

III. HyperD Definitions and Structure

The following notations and definitions will be used throughout this paper:

Let \( D = (V_D, E_D) \) be a graph representing a HyperD instance with \( d = |V_D| \) nodes and \( e = |E_D| \) edges. \( D \) is taken to be a subgraph of the \( n \)-dimensional hypercube \( H = (V_H, E_H) \) with \( V_D \subseteq V_H, E_D \subseteq E_H \). Clearly, \( d \leq 2^n \) and \( e \leq n2^{n-1} \).

Definition 3.1 (Full and Partial Hypercube): \( D \) is a full hypercube if \( V_D = V_H \) and \( E_D = E_H \). Otherwise \( D \) is a partial hypercube.

Definition 3.2 (Label Space): The Label Space \( LS(H) \) is the set of binary labels of a hypercube \( H \). If \( H \) is an \( n \)-dimensional hypercube, \( |LS(H)| = 2^n \), the number of nodes in \( H \). The Label Space \( LS(v) \) of a node \( v \in D \) is the set of binary labels assigned to \( v \). Since a node in a partial hypercube \( D \) may have to manage several labels to allow for simulating all the paths in a full hypercube, \( LS(D) = LS(H) \).

HyperD has the following properties:

- HyperD is a set of interconnected peers forming in an ad-hoc fashion a single network with no centralized control.
- Nodes in HyperD establish bidirectional connections which correspond to edges in the graph \( D \) modeling the network. Thus \( D \) is an undirected graph.
- HyperD is dynamic and nodes may enter or leave the network at any time. Connections in a HyperD are rearranged to deal with such changes in the network.
- HyperD approximates an \( n \)-dimensional hypercube if the number of nodes is between \( 2^{n-1} \) and \( 2^n \).
- Each node \( v \in \text{HyperD} \) is assigned a subset of binary labels \( LS(v) \subseteq LS(H) \) where \( |LS(v)| \geq 1 \).
- Each label \( l \in LS(H) \) is assigned to a unique node in HyperD.
- Each node \( v \in \text{HyperD} \) knows the label space for each of its neighbors.
- If the number of nodes exceeds \( 2^n \), HyperD expands to a virtual \((n+1)\)-dimensional hypercube.

Figure 1 shows an example of a HyperD of 10 nodes. During the existence of this network 13 nodes have entered, 3 of which have at some point left the network. The remaining 10 nodes form a partial (incomplete) 4-dimensional HyperD. \( u \) is a node with label space \( LS(u) \). A solid line denotes an edge joining two nodes in HyperD, and a dotted line represents a vacant edge in the underlying hypercube structure. We will use this example to illustrate some of the algorithms in this paper.

IV. Communication in HyperD

In a dynamic distributed network such as a DDFD, available resources such as bandwidth and processing power may be limited. To reduce the network load and increase throughput it is important to reduce the cost of communication between nodes. Broadcasting and pairwise communication are the most frequent activities in a distributed network. In this section we adopt known hypercube communication algorithms to implement a unicast (a node sends a message to a single node) and a broadcast (a single node sends a message to all other nodes) in a HyperD.

Unicast First we present a routing algorithm for HyperD. A path between two nodes is determined by their respective label spaces. Details are summarized in Algorithm 1. Some of the definitions used in this algorithm are listed below:

Definition 4.1 (Neighborhood): Two distinct nodes in a full hypercube are adjacent if their labels differ in exactly one position. In this case we also speak of the labels being
Algorithm 1 HyperDRouting(D, s, d)

Input: HyperD Network D, Source Node s ∈ V_D, Destination Node d ∈ V_D.
Output: A path P from s to d, where P ⊆ E_D.
1: P ← ∅
2: if d ∈ neighborhood(s) then
3: P ← P ∪ {edge(s, d)}
4: else
5: Select a node t ∈ neighborhood(s) such that
6: min(distance(t, d)). Ties are broken randomly.
7: P ← P ∪ HyperDRouting(N, t, d)
8: end if
9: return P

Algorithm 2 HypercubeBroadcast(H, s, c)

Input: HyperD D, Source Node s ∈ V_H, Current Node c ∈ V_H.
Output: A broadcast (spanning) tree B, where B ⊆ E_H.
1: B ← ∅
2: k ← dimension of H
3: dim ← dimension of neighbor s
4: if s = c then
5: for i = k − 1 to 0 do
6: B ← B ∪ {edge(c, neighbor_i(c))} ∪ HypercubeBroadcast(H, c, neighbor_i(c))
7: end for
8: else
9: for j = k − 1 to dim + 1 do
10: B ← B ∪ {edge(c, neighbor_j(c))} ∪ HypercubeBroadcast(H, c, neighbor_j(c))
11: end for
12: end if
13: return B

Algorithm 2 can be summarized as follows: The broadcasting node sends the message to all its neighbors (Steps 4-7). In turn, each node v upon receiving a message from its i-dimensional neighbor u, forwards the message to all of its k-neighbors where i < k < n (Steps 8-12). The algorithm returns a spanning tree with root at the source node (Step 13).

An example where Algorithm 2 is used to broadcast a message in a 4-dimensional hypercube originated at node s is illustrated in figure 2.

Fig. 2: Example of a broadcast in a 4-dimensional hypercube

We adopt this approach to fit the specific characteristics of a HyperD. Algorithm 3 shows the steps used to broadcast a message in a HyperD network. We have to consider the fact that each node in HyperD may own more than one hypercube label.

Algorithm 3 is used to identify one of the several possible spanning trees with root at the broadcasting node. We offer additional comments here for a better understanding of the algorithm. The broadcasting node s selects a label l_s ∈ LS(s) and sends the message to all its neighbors relative to l_s (those neighbors having labels adjacent to l_s). Each message includes the chosen label l_s. The broadcasting node would invoke
Algorithm 3 HyperDBroadcast$(D, s, c, l_s, l_c)$

**Input:** HyperD Network $D$, Source Node $s \in V_D$. Current Node $c \in V_D$, Label $l_s \in s$, Label $l_c \in c$.

**Output:** A broadcast (spanning) tree $B$, where $B \subseteq E_D$.

1: $B \leftarrow \emptyset$
2: $dim \leftarrow \text{dimension}(l_s, l_c)$
3: $k \leftarrow \text{dimension of } D$
4: for each node $N_i \in \text{neighborhood}(c) \neq s$ do
   5: if $\text{difference}(l_c, l_i) \in N_i) == 1 \&\& \text{dimension}(l_c, l_i) > dim$ then
   6: $B \leftarrow B \cup \{\text{edge}(c, N_i)\} \cup \text{HyperDBroadcast}(D, c, N_i, l_c, l_i)$
   7: end if
   8: end for
9: for each label $l_i \in c$ do
10: if $\text{difference}(l_c, l_i) == 1$ then
11: $B \leftarrow B \cup \text{HyperDBroadcast}(D, c, c, l_c, l_i)$
12: end if
13: end for
14: return $B$

HyperDBroadcast$(D, s, c, l_s, l_c)$ to initiate the broadcast. Recursively, if a node $c$ receives a message from its neighbor $s$ (including label $l_s$), it forwards the message to all its $\text{neighborhood}(c)$ where $i < k < n$ including label $l_c$. (Steps 4-8). Note that a node may have more than one label adjacent to $l_s$. If $c$ itself has a label $l_w$ that is a $j$-neighbor of $l_c$ where $j > k$, it sends a message to all its neighbors who have a label that is a $g$-$\text{neighbor}$ of $l_w$ where $j < g < n$ including label $l_w$ (Steps 9-13).

Figure 3 shows an example of a broadcast in a 4-dimensional HyperD (from figure 1).

![Fig. 3: Example of a broadcast in a 4-dimensional HyperD](image)

V. HYPERD MAINTENANCE

A. Node Enters

When a new node $u$ intends to enter a HyperD network it first needs to connect some existing node $n_0 \in \text{HyperD}$. In turn $n_0$ broadcasts the enter request to all other nodes in the network. The broadcast propagates along a hypercube minimum spanning tree with root at $n_0$. When a node $n_i$ receives the enter request it responds to the broadcasting node if $|LS(n_i)| > 1$ (it has an available label which can be assigned to the new node). Since the new node only needs to discover a single available label, if $n_i$ responds to the broadcast it does not forward further the enter request to other nodes. This limits the broadcast range, thus saving valuable network resources. Algorithm 4 is used for an enter broadcast.

Algorithm 4 HyperDEnterBroadcast$(D, s, c, l_s, l_c)$

**Input:** HyperD Network $D$, Source Node $s \in V_D$. Current Node $c \in V_D$, Label $l_s \in s$, Label $l_c \in c$.

**Output:** Set $A \subseteq V_D$ of all nodes with more than one label.

1: $A \leftarrow \emptyset$
2: $dim \leftarrow \text{dimension}(l_s, l_c)$
3: $k \leftarrow \text{dimension of } D$
4: if $|LS(c)| > 1$ then
   5: $A \leftarrow A \cup \{c\}$
   6: else
   7: for each node $N_i \in \text{neighborhood}(c)$ do
   8: if $\text{difference}(l_c, l_i) \in N_i) == 1 \&\& \text{dimension}(l_c, l_i) > dim$ then
   9: $A \leftarrow A \cup \text{HyperDEnterBroadcast}(D, c, N_i, l_c, l_i)$
   10: end if
   11: end for
   12: end if
13: return $A$

Algorithm 4 can be summarized as follows: If the broadcasting node has itself available labels there is no need to forward the request and the broadcast will end (Steps 4-5). Otherwise the node will forward the message to all its neighbors which recursively will repeat the same process (Steps 6-12). The algorithm will return to the broadcasting node a set of nodes $A$ which have available labels. If HyperD is a partial hypercube the new node will receive at least one label offer, $|A| \geq 1$. Otherwise if HyperD is a full hypercube (no available labels) the new node receives no labels, therefore $|A| = 0$. In the event $|A| = 0$, HyperD has to expand to the next dimension to allow the new node to enter. Each node in the network will have to double its label space by padding each label with a 1 and 0 at its highest dimension to create this way two distinct labels.

![Fig. 4: Example of a enter broadcast in a 4-D HyperD](image)
Note that only 8 of the 10 nodes are reached by this broadcast.

Algorithm 5 HyperDEnter(D, n, v)

Input: HyperD D, New node n \not\in V_D, Node v \in V_D.
Output: A new HyperD D including the new node n.
1: //Node n contacts node v \in V_D
2: L \leftarrow HyperDEnterBroadcast(D, v, l_v, l_v)
3: if L = \emptyset then
4:     Expand D to the next dimension
5:     L \leftarrow HyperDEnterBroadcast(D, v, l_v, l_v)
6: end if
7: Node a \leftarrow Null
8: for each node i \in L do
9:     if distance(i, v) < distance(a, v) then
10:         a \leftarrow i
11: end if
12: end for
13: Node a assigns a label to n
14: for each node k \in neighborhood(a) do
15:     if distance(n, k) == 1 then
16:         E_D \leftarrow E_D \cup edge(n, k)
17:     end if
18:     if distance(a, k) > 1 then
19:         E_D \leftarrow E_D - edge(a, k)
20: end if
21: end for
22: return D

The new node enter broadcast is the most important step in the enter procedure which is shown in Algorithm 5. The following definition is needed here.

Definition 5.1 (Internal Edges): Two labels are said to be joined by an internal edge if the two labels are adjacent and both belong to the same node. The internal degree of a label is the number of internal edges incident to the label.

The new node n identifies a node v from the HyperD network (Step 1). Node v then broadcasts the enter request. The list of nodes with available labels returned by the broadcast are stored in L (Steps 2-6). The new node then selects a node a from L which is the closest in the virtual network as its first neighbor. (Steps 7-12). Choosing the closest node helps maintain a good alignment between the virtual network and the physical network. Node a then donates one of its labels to the new node n (Step 13). Node a chooses the label to donate using the following criteria: a) the label with the smallest internal degree. b) Ties are broken by selecting the label with the largest binary value. Upon receiving the label the new node connects to its new neighbors (Steps 15-17). The donor node a disconnects from those neighbors relative to the donated label (Steps 18-20).

Figure 5 shows an example of a node entering a full 2-dimensional HyperD. Node [1] broadcasts an enter request. Since the network is a full hypercube (figure 5a), it needs to expand to a 3D hypercube. Assume node [2] was the first node contacted by [3]. Node [4] donates its largest binary label 100 to [5]. Node [5] then connects to nodes [6] and [7]. The final result is shown in figure 5b.

Figure 6 shows an example of a node entering a partial 3D HyperD. Node [8] broadcasts an enter request and receives a response from nodes [9] and [10] which have available labels in their label space (figure 6a). Assume node [11] accepts a label from node [12]. [12] chooses label 110 since it has the lowest internal degree and its binary value is greater than 011. [11] connects to [13], [14] and [15]. Node [12] disconnects from [11]. The final result is shown in figure 6b.

B. Node Departures

A departing node n may announce its departure and inform all neighboring nodes or may fail or simply not announce its departure. In the latter case the departure is detected during any periodical connection check or during any network activity that involves the departed node. Algorithm 6 describes the steps that are taken when the departure of a node is detected.

In Algorithm 6 once the departure of n is announced (or detected), the label l with the largest value is selected from the label space of n (Steps 3-7). Node k which owns the label adjacent at the lowest dimension to l is selected to take over the label space of n (Steps 8-10). Node k will then connect to all former neighbors of n (Steps 11-13).

Figure 7 shows an example of a node departing HyperD. Node [18] leaves the HyperD network shown in figure 7a. Node [19] assumes control of the label space of [18] since [18] owns label 110 which is adjacent at position 0 to the largest label.
Algorithm 6 HyperDDeparture(D, n)

Input: HyperD Network D, A departing node n ∈ V_D.
Output: A new HyperD D without the departing node n.
1: Label l ← −1
2: //Departure of node n is announced or detected
3: for each label l_i ∈ LS(n) do
4:   if l_i > l then
5:     l ← l_i
6:   end if
7: end for
8: Choose l_k|dimension(l, l_k) == 0
9: Choose node k|l_k ∈ LS(k)
10: LS(k) ← LS(k) ∪ LS(n)
11: for each node u ∈ neighborhood(n) do
12:    E_D ← E_D ∪ edge(k, u)
13: end for
14: return D

Fig. 7: Example of a node departing a HyperD

from 2111. Node 11 only needs to connect to 33 since it is already connected to the other former neighbors of 133. The final result is shown in figure 7b.

C. Periodical Maintenance

During the existence of HyperD, nodes and links may fail. To correctly process a query, each node needs to maintain updated information about its neighbors. To accomplish this goal, periodical connection check messages are exchanged over all the links in HyperD. The total network bandwidth utilization to perform this maintenance depends on the number of links as well as the frequency of these checks. In a hypercube network the number of links per node grows with the size of the network as compared with a preferential attachment random network where the average number of links per node is constant. Therefore it is more expensive to perform this task in a HyperD when the number of nodes increases. To minimize the cost we can reduce the frequency with which such checks are performed. This is particularly feasible if the network is relatively stable. In addition, some of the network changes can be detected during any query activity.

VI. EXPERIMENTAL EVALUATION

In this section we summarize some of the experimental results aimed at comparing a GaianDB which grows randomly using a process of preferential attachment (GaianDB_PA) and a GaianDB which grows following the HyperD protocol (GaianDB_H). The performance of GaianDB_PA has been extensively evaluated via experiments. Some of the results have been summarized in [3]. In addition to experimental results a number of coarse grained query cost models for DDFDs have been considered [17] to identify those conditions where a particular topology performs better.

A. Experimental Setup

The Test environment is shown in Figure 8. The environment comprises 12 IBM Blade Servers distributed over two Blade Centers. These are used to host the GaianDB nodes. An additional Blade Server was used to coordinate the growth of the network of GaianDB nodes, to inject queries, and to measure the performance of the resulting distributed database network. These machines were connected by a Gigabit Ethernet LAN as shown in Figure 8.

Fig. 8: Experimental test harness comprising 12 blades

"Blade Center 1" comprised 9 Blade Servers each having 2 hyper-threaded 3200 GHz processors and 5 GB of RAM. "Blade Center 3" comprised 13 Blade Servers, each of which have 2 hyper-threaded 3800 GHz processors and 5 GB of RAM. The Operating System installed on all blades is Red Hat Enterprise Linux ES release 3.

Each Server in the Test Environment is capable of running a number of GaianDB. Each node runs in its own Java Virtual Machine (JVM) with 128 Mb of memory allocated to each JVM and each configured on a separate TCP/IP port. GaianDB’s of various sizes were grown using the Preferential Attachment connection method described in [2] and separately grown using the HyperD connection method described in this paper. The common utility was used to report the network traffic on each server. This network data was aggregated to determine the total amount of data transmitted during the experiments.

B. Experimental Results

The first experiment evaluated the cost of growing, and maintaining Gaian_PA and GaianBDH networks. The method was to start a number of GaianDB nodes, measure the network load while starting and then for a period afterwards to assess the "background load" of network maintenance. Then we start a number of additional nodes to create a bigger network, etc. This was performed with network sizes staged at 2, 4, 8, 16, 32, 48 and 64 nodes. Figure 9 shows the average network load for each stage of network activity - starting nodes...
and monitoring the background load respectively. The results for \(Gaian_{PA}\) and \(Gaian_{BDH}\) are shown together to allow for easy comparison.

![Fig. 9: Bandwidth utilization for growing and monitoring different GaianDB networks](image)

It can be seen that the enter costs and maintenance cost of a \(Gaian_{DBH}\) increases at a faster rate than that of a \(Gaian_{DBPA}\). Traffic is about equal at 8 nodes and beyond that, \(Gaian_{DBH}\) takes more bandwidth to maintain. It should be noted that the rate with which nodes checked connections was configured to be relatively frequent - once every 5 seconds. This will accentuate the effect of background connection checking and in real life networks this frequency, can be reduced.

The second experiment evaluated the costs of broadcasting a query in a \(Gaian_{DBH}\) using flooding, and hypercube routing to propagate the queries. Figure 10 shows the relative costs of propagating queries. It can be seen that cost of the hypercube routing algorithm is less than half of the flooding algorithm and the difference increases with the size of the network. The queries issued here return a one row of data per node. If we are returning larger amounts of data, the difference in bandwidth utilization will become amortized.

![Fig. 10: Network traffic/query for different routing protocols](image)

![Fig. 11: Query time for different routing protocols](image)

Figure 11 shows the query times associated with the different routing protocols and it is interesting to note that the query times are roughly the same. This is true in a network that is not constrained on bandwidth, however in a network constrained by bandwidth, the lower network load of the hypercube routing protocol should result in greater throughput, and lower latency for hypercube broadcasted queries.

In any distributed network the total network load is a function of several parameters. A certain combination of values for these parameters may favor a particular topology over others. To estimate those conditions in which a hypercube based topology performs better than a random topology using a preferential attachment protocol we select only dominant factors and some representative values.

Let \(RC\) be the rate of topology changes (node enters and departures per minute), let \(CC\) be the average estimated cost of a single topology change (in kb), let \(RQ\) be the rate of queries issued each minute, let \(CQ\) be the average cost of a single query (in kb) and let \(BC\) be the background cost of periodically checking connections (once each minute). Then the total bandwidth utilization per minute can be estimated as follows: 
\[
\text{Total} = CQ \times RQ + CC \times RC + BC.
\]

Note that \(CQ, CC\) and \(BC\) are all functions of the size of the network \(n\). We then compare the two topologies by selecting estimated values for some of the parameters for a network of approximately 64 nodes. We obtain the following functions for each topology:

- \(Gaian_{DBH}\): \(Total_H = 35 \times RQ + 300 \times RC + 855\).
- \(Gaian_{DBPA}\): \(Total_{PA} = 165 \times RQ + 100 \times RC + 200\).

The comparison results are shown in figure 12. It can be seen that a higher rate of queries favors the hypercube topology. Only in those cases where the network changes are the domi-
nating activity will the hypercube network be more expensive.

![Fig. 12: 3D total network traffic comparison graph](image)

VII. CONCLUSIONS AND FUTURE WORK

In this paper we show that building a hypercube based DDFD is not only better but it proves to be beneficial in terms of reducing the network traffic if the rate of queries is significantly higher than the rate of topology changes. We summarize our observations and list some of the advantages and disadvantages of using HyperD as compared to a random scale free network.

**Advantages:** Queries are performed more efficiently; routes can easily be identified between nodes. Message forwarding is only done using local information and node labels; alternative routes are easier to find; it is easier to parallel process queries; the network is denser thus more robust against fragmentation.

**Disadvantages:** There is an additional overhead to maintain the hypercube structure; maintaining the integrity of the structure under constant and concurrent changes may be complex and prone to errors; the network is denser, increasing the cost of maintaining connections.

Possible directions for future research may include the following: Explore potential benefits of allowing the network to operate at different hypercube dimensions; identify additional ways to reduce physical/logical misalignment; avoid errors and network fragmentation caused by frequent and concurrent node departures/enters; compare the cost of more complex queries such as joins and semijoins; and extend the experimental results to large networks.

VIII. ACKNOWLEDGMENTS

Research was sponsored by the U.S. Army Research Laboratory and the U.K Ministry of Defense and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defense or the U.K Government. The U.S. and U.K. Governments are authorized to reproduce and distribute for Government purposes notwithstanding any copyright notation hereon.

REFERENCES


