Replica Placement in a Dynamic Network*

Gurdip Singh and Mahesh Bommareddy
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Abstract: We study the problem of placement of replicas of a database (or a shared resource) in a dynamic network. We develop a set of protocols that maintain a path from each site to its nearest replica such that the cost of accessing the nearest replica is below a certain threshold. The protocols determine the number of replicas needed and the sites where these replicas must be placed. This is useful in read-intensive applications which impose time constraints on read operations. The protocols reconfigure the placement in response to changes in link costs. This may involve recomputing paths and relocating, adding or removing replicas.

1 Introduction

Many distributed applications require different processors in a system to access a shared database (or a shared resource). For example, in an airline reservation system, several transactions for reservation/cancellation may be concurrently active trying to access shared data, namely the number of seats already reserved. For reasons of performance and fault tolerance, several copies (replicas) of the shared database (or the resource) may have to be maintained. In this case, each site can access the nearest replica which can result in significant performance advantage over a single replica system. Furthermore, failure of any site containing a replica will not block the network.

In many applications such as distributed dictionary and mapping service [1] [5], the number of read operations may outnumber the write operations. In addition, some applications may require fast response time for read operations to ensure some real-time constraints or increase the transaction throughput. From the performance view-point of read operations, a replica must be maintained at each site. However, as the number of replicas is increased, the cost of a write operation increases. Thus, the choice of the number of replicas and set of sites where replicas have to be placed is crucial. In this paper, we study the problem of placing a minimum number of replicas such that the communication cost of read operations is below a certain threshold, *Cost*. This would satisfy the time constraints of the read operations and at the same time, the cost of a write operation will not increase without bounds (since we maintain a minimum number of replicas to satisfy the constraints). In addition, we assume that link costs can change, and therefore, the placement may have to be changed dynamically to meet the constraints.

Maintaining a minimum number of replicas in an arbitrary topology is an NP-complete problem [4]. Therefore, we start with an arbitrary initial placement of replicas in the network (for example, in a large network, we may start with a replica in each subnetwork). We propose a protocol *Path_To_Nearest* which constructs a path from each site to its nearest replica (this path is then used to access the nearest replica). The protocol is a modification of the protocol in [6] which constructs paths to a single site. In general, the set of shortest paths will form a forest, where each subtree in the forest is rooted at a replica site. We convert this forest into a rooted tree by introducing some additional edges. This tree is then used for subsequent placement decisions.

We develop a protocol, *Increase_Cost*, which is invoked whenever an edge cost increases. An increase in the cost of a link may invalidate the cost constraint for some nodes (*i.e.*, the current nearest replica may not remain within distance *Cost*). In this case, we try to construct a path to another replica using only the

*This work was supported by NSF Research Initiation Award CCR-9211621.

1063-6927/94 $03.00 © 1994 IEEE

edges of the tree. Thus, nodes from one subtree of the original forest may join another subtree. If this is not possible, then the protocol tries to relocate replicas along the edges of the tree to satisfy the cost constraints (the replicas are moved towards the nodes with longer paths). If this also fails, a new replica is introduced.

To deal with decreases in link costs, we propose a protocol, *Decrease_Cost*. The goal of this protocol is to recompute paths and relocate replicas so that the number of replicas can be reduced. If no link cost changes happen for a sufficient period of time, the system will converge to a minimum number of replicas with respect to the tree. All reconfigurations are local in nature and only nodes in the vicinity of the link whose cost has changed participate in the protocol. This locality property allows concurrent reconfigurations to proceed in different parts of the network and hence, increase efficiency.

Thus, our system consists of three main components: *Path_to_Nearest*, *Increase_Cost* and *Decrease_Cost*. The initial placement may not satisfy the constraints and we may have to initiate *Increase_Cost* (in case there exists a node with distance greater than *Cost* from the nearest replica) or *Decrease_Cost* which will eliminate unnecessary replicas in this placement. Since all network edges are not used in making placement decisions, it may not result in an optimal placement if the entire network is considered. However, we feel that our solution is a reasonable approximation since the tree edges correspond to least cost paths from nodes to the replicas. It may happen that due to increase in the cost of the tree edges, the tree may no longer correspond to shortest paths. It would be expensive to recompute shortest paths after every link cost change. Instead, it would be advantageous to recompute the tree to include shortest paths if the number of replicas required to meet the constraints in the existing tree exceeds a certain number (which would indicate the cost of the tree edges has increased considerably). The protocols have been tested using discrete event simulation.

[7] proposed an algorithm for placement of replicas in a tree topology. The placement is dependent on the read-write pattern in the network and does not take edge costs into account. Thus, more replicas are introduced near sites which invoke more read operations. In our system, the frequency of operations can be taken into account by modifying the edge costs appropriately. For example, if a node associates higher costs with edges (in addition to the actual cost of the edges) over which it has received more read requests then this will either result in relocation of replicas towards those edges or placement of a new replica near them.

The paper is organized as follows. In the next section, we discuss our model of distributed computation. Section 3 discusses the protocol *Path_to_Nearest*. In Section 4, we present the protocol *Increase_Cost* to deal with increase in link costs and in Section 5, we present *Decrease_Cost* which is invoked when a link cost decreases. Section 6 presents the conclusion.

2 Model

We model a network as an undirected graph with N nodes and E edges, where nodes represent the processors and edges represent the communication links. Each edge (i, j) has a positive weight, $w(i, j)$, assigned to it. This weight represents the cost of using that link and this cost may change with time. We assume that messages are not lost and they arrive at their destination in the order sent within finite but unpredictable time. Initially, each node knows the weights of all edges incident on it. Initially, the set of nodes is divided into two subsets, *replica sites* and *ordinary sites*. One site among the replica sites is designated as the *sink*.

3 Constructing Shortest Paths

We propose a protocol, *Path_to_Nearest*, whose goal is to obtain a rooted tree from the given arbitrary topology. Let $path(i)$ denote the path from i to its nearest replica. In general, the graph G obtained by collecting $path(i)$ for each node i will be a forest. For example, Figure 1 shows an arbitrary topology with initial placement of replicas $r1$, $r2$, $r3$ and $r4$. Figure 2 gives a forest obtained by collecting $path(i)$ for each site i. The protocol *Path_to_Nearest* obtains the graph G and then adds some edges to convert G into a rooted tree with *sink* as the root. Figure 3 gives a possible rooted tree obtained from the forest in Figure 2. The resulting tree has the following properties:

- Every node has a path in the tree to its nearest replica which is the shortest path to this replica.

- Every node knows the neighbor through which the nearest replica site can be reached.
- Every node knows the neighbor through which the root can be reached.

The protocol has two phases. The first phase creates a forest and the second phase converts it into a tree.

3.1 Constructing a forest

We will first assume that there are no topological changes (*i.e.*, changes in edge costs) during the execution of the protocol. Extension to handle topological changes is discussed in [2]. Each node i maintains the following set of variables:

- $parent_i$ denotes i's parent in the tree.
- $rep[i,k]$ denotes the nearest replica reachable via neighbor k.
- $dist[i,k]$ denotes the distance to $rep[i,k]$ via k.
- min_rep_i denotes the nearest replica.
- $dist_rep_i$ denotes the distance to min_rep_i.
- $pref_i$ points to the node which leads to min_rep_i.

This phase is initiated by the *sink* and proceeds in a sequence of cycles. In each cycle, distance information is accumulated at each node and is used to obtain a better estimate of the nearest replica. If the length of $path(i)$ is k, then at the end cycle k, i will have correct values in its variables.

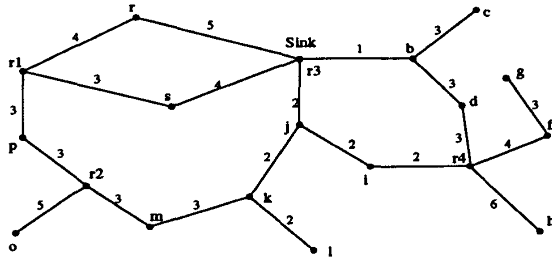

Figure 1: An arbitrary topology.

Each cycle consists of two phases: a downtree phase in which information flows from the *sink* to all other nodes and an uptree phase, in which messages are sent towards the sink. In the downtree phase, the *sink* sends a message $MSG(cycle, distance, sink)$ to its neighbors with $distance = 0$. In general, when MSG is sent by i, its first argument contains the cycle number, the third argument contains the identity of the nearest replica and the second argument is the distance to this replica. When node i receives

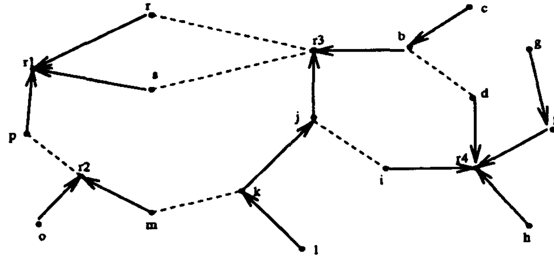

Figure 2: A forest of trees.

a message $MSG(cycle, distance, replica)$ for the first time from site k, it does the following: $parent_i := k$; $rep[i,k] = replica$ and $dist[i,k] = distance + w(i,k)$. Then, it sends $MSG(dist_rep_i + w(i,k), min_rep_i)$ to all neighbors except k.

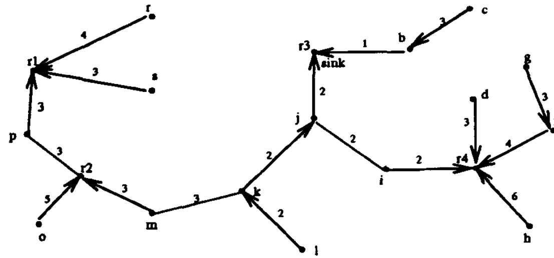

Figure 3: A single rooted tree.

To generate the messages in the uptree phase, each node i waits for a message from each neighbor except $parent_i$. On receiving the message $MSG(cycle, distance, replica)$ from neighbor k, i does the following: it sets $dist[i,k] = distance + w(i,k)$ and $rep[i,k] = replica$. After receiving a message from each neighbor, i computes the nearest replica according to the information received so far as follows: it sets $dist_rep = min(dist[i,k] : k \in N_i)$, $min_rep_i = rep[i,k]$ and $parent_nearest_i = k$ such that $dist_rep_i = dist[i,k]$. Then, it sends a message $MSG(cycle, dist_rep_i, min_rep_i)$ to $parent_i$. Thus, during the uptree phase, nodes use the accumulated distance information to obtain a better estimate of the nearest replica. When *sink* receives a response from each neighbor, it starts the next phase.

In each cycle, each node i uses a variable $change_i$ to keep track of whether it obtained distance information which resulted in a change in min_rep_i or $dist_rep_i$. If any node i sets $change_i$ to true, the *sink* is notified

during the uptree phase of this fact so that it can initiate the next cycle. Figure 2 shows the forest obtained for the network in Figure 1.

3.2 Constructing a Tree

In general, the collection of paths obtained by the first phase of *Path_to_Nearest* may be a forest. In this section, we will describe the second phase which converts the forest into a rooted tree. In this phase, the tree rooted at the *sink* tries to absorb all other trees so that on termination, only one tree (which includes all nodes) rooted at the *sink* remains. To initiate this phase, the root sends a TREE message to its neighbors. This message carries a list, *r_list*, of all the replicas whose trees have already been absorbed (initially, $r_list = \{sink\}$). When a node i receives a TREE message from j, it behaves as follows: It first sets $parent_i$ as $pref_i$. If i has already received a TREE message or min_rep_i is in *r_list* then it sends a N_ACK message to j. Otherwise, i requests permission from min_rep_i to merge with $j's$ tree. To request permission, it sends ADD_REQ to $pref_i$, which is further propagated along the tree edges until it reaches min_rep_i. When the replica r receives an ADD_REQ message, it behaves as follows: If this is the first such message received, then it sets $parent_r$ to the node from which this message is received and sends ADD_GRANTED message to this node. On receiving the ADD_GRANTED message, a node j sets $parent_j$ to the node from which ADD_REQ was received and propagates ADD_GRANTED to this node. Finally, the message reaches the node, i, which originated ADD_REQ. This node designates the node from which the TREE message was received as $parent_i$. To ensure that every node receives the TREE message, each node propagates this message to all neighbors (except to which is send ADD_REQ) whenever it learns of the initiation of this algorithm.

Figure 3 shows a rooted root for the forest in Figure 2. Note that the arrows in the figure indicate the parent pointers (and $pref_i$ may be different from $parent_i$). The propagation of TREE messages will ensure that eventually a single rooted tree is formed. However, the sink has to be informed that this process has terminated. For this purpose, we propagate acknowledgements to the root. The propagation is similar to the protocol in [3]. In addition, during this propagation, we also collect some information required for subsequent maintenance of the replicas. Let i and j be neighboring nodes. Then, we define $depth(i, j)$ as the distance of the farthest node k from i such that i, j and k have the same nearest replica, say r, and j is on the shortest path from k to r. For example, in Figure 3, $depth(j, k) = 4$ because l is at distance 4 from j and the shortest path from l to $r3$ is through j. For each pair of neighboring nodes i and j such that they have the same nearest replica, we will compute $depth(i, j)$. Let max_depth_i denote the maximum of $depth(i, j)$ for all neighbors j.

The propagation of the acknowledgments is initiated by the leaves of the newly formed tree. After a node i finds that it is a leaf then it sets $parent_i$ to $pref_i$ and max_depth_i to 0. Then it sends an ACK(max_depth_i) message to $parent_i$. Each site i waits for an ACK from each child except $parent_i$, and computes the depth information. Then, it sends an ACK(max_depth_i) message to its parent. When the root receives an ACK message from each neighbor, it knows that the tree has been formed and all nodes have the depth information.

4 Increase in an edge cost

In this section, we discuss the reconfiguration required in response to an increase in link cost. As a result of an increase in the link cost, a node i may no longer be at the required distance from its previous nearest replica. For example, in Figure 4, if the cost of link $(r1, p)$ changes from 3 to 8 and $Cost = 7$ then p will be at a distance greater than $Cost$ from $r1$. However, $r2$ is still within $Cost$ distance from p. We first attempt recomputation of paths to check if a node can connect to another replica and satisfy the cost constraint. A node which is not able to find a replica within $Cost$ distance is called an *orphan*. For example, if the cost of (k, j) in Figure 4 changes from 2 to 5 then after recomputation of paths, as shown in Figure 5, l will still be at distance 8 from the nearest replica. If orphan nodes exist, then we attempt to relocate the replicas so that cost constraints are satisfied. Figure 6 shows a possible relocation to satisfy the cost constraints, where the replica at the sink node is moved to j. Finally, if relocation is not possible, we introduce new replicas. We will now discuss the various components of the protocol *Increase_Cost*.

4.1 Recomputation of Paths

Let there be an increase in the cost of edge (u, v). If (u, v) is not a tree edge or if the nearest replicas

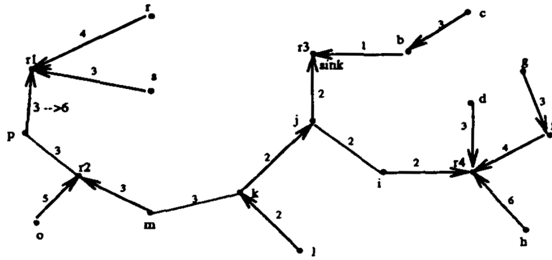

Figure 4: A rooted tree.

of u and v are different then this change will not invalidate the cost constraint or change shortest paths. Hence, we will assume that (u, v) is a tree edge and both nodes have the same nearest replica. Assume that $pref_u = v$ (from now on, we will refer to u as the *victim* node). Let δ be the change in the link cost. If $\delta + max_depth_u \leq Cost$ then this change does not invalidate the cost constraint at any node, and u simply updates $dist_to_rep_u$. For example, if the cost of link $(r1, p)$ in Figure 4 changes from 3 to 6, where *Cost* = 7, then all nodes will still be within *Cost* distance from their nearest replica. This will, however, invalidate the depth information at nodes in the path from u to min_rep_u. To update this information, u initiates a procedure *Change_depth* (discussed in the next section).

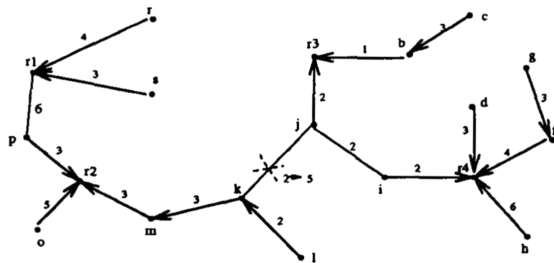

Figure 5: Illustration of orphan nodes

Otherwise, $\delta + max_depth_u > Cost$. Let S denote the set of sites i such that u is on the shortest path of i to its nearest replica (S also includes u). Then, all nodes in S will incur an increase in distance from the nearest replica, and for some node(s) i in S, min_rep_i is at distance greater than *Cost*. For instance, in Figure 4, if the cost of $(r1, p)$ had increased to 8 then node p will be at a distance greater than *Cost*.

Let Rep_set_i denote the set of replicas r such that there exists a path from i to r which does not include any other replica. For example, Rep_set_j in Figure 3 is $\{r2, r3, r4\}$. It might be possible that some replica in Rep_set_i is at distance less than or equal to *Cost* from i. To determine this, each site i in S obtains its distance to replicas in Rep_Set_i (note that *Rep_set* is identical for all nodes in S). For this purpose, u sends a message INCREASE(δ) to its children (including itself) which is further propagated to all nodes in S.

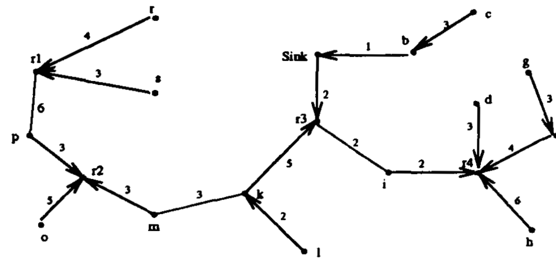

Figure 6: Relocation of Replicas

When a node i receives the message INCREASE(δ), it behaves as follows: It sends a message REQ_DIST to each neighbor j in the tree that is not in S and INCREASE(δ) to neighbors in S. On receiving REQ_DIST, a node $j \notin S$ responds with a REQ_ACK($dist_to_rep_j, min_rep_j$) message. On receiving a REQ_ACK(l, r) message from j, node i behaves as follows: Node i remembers that its distance to replica r is $l + w(i, j)$. In addition, it sends a message REQ_ACK($l + w(i, k), r$) to each neighbor k in S (so that other nodes in S can determine their distance to r). When node k receives the message REQ_ACK(p,r), it further sends the message REQ_ACK($l + w(i, x), r$) to each neighbor x in S. Thus, each node i in S will know the distance to each replica in *Rep_set*. Then it can choose the replica nearest to it as min_rep_i and sets $parent_i$ to j, where j is the node from which the message carrying minimum distance was received. For example, in Figure 4, if cost of $(r1, p)$ increases to 8, then paths will be recomputed as shown in Figure 5. In this case, p will connect to $r2$ via edge $(p, r2)$. As a result of this recomputation, the depth information may get invalidated. Therefore, each node i in S initiates *Change_Depth* to update the depth information.

4.2 Scanning for Orphan Nodes

than *Cost*. After recomputation of paths is done, it is possible that there exists nodes in S which are *orphans*. To scan for orphan nodes, *victim* sends a message REPORT to all neighbors in S. Each node i remembers the node, rec_i, from which it had received the REPORT message. If i has no neighbor other than rec_i which belongs to S then it sends a REPORT_ACK message to rec_i. If node i has $dist_rep_i > Cost$, it sends $w(i, rec_i)$ in the REPORT_ACK message. Otherwise, distance zero is sent in this message.

Node i waits until it receives REPORT_ACK from all other neighbors to which it sent a REPORT message. After receiving these messages, i determines the maximum orphan distance, *max*. If $max = 0$ then it sends 0 in the REPORT_ACK message; otherwise, it sends $max + w(i, rec_i)$ in the message. Also, every node, including the victim node, that receives the REPORT_ACK from its neighbors, maintains the maximum orphan distance along all of its branches except $parent_i$. Eventually, the victim node which has initiated this scan receives REPORT_ACK from all nodes to which it had sent REPORT. At this point the victim node can check if there are any orphan nodes. If it has received a non zero value from a node j, then there exists an orphan node reachable via j.

4.3 Relocation of Replicas

On detecting an orphan node, we attempt to relocate the replicas. We attempt relocation of only those replicas which are on the path from the victim to the sink. Since we ensure that the placement of replicas is such that they are as close to the root as possible, it is sufficient to check only this set of replicas for relocation (relocation of any other replica will not help in satisfying the cost constraint). For example, consider the network shown in Figure 4. Let there be a change in cost in the link (j, k) from 2 to 5. Because of this link change, shortest paths to replicas are recomputed and the resulting connections are as shown in Figure 5. Then l will be an orphan (the path $l \rightarrow k \rightarrow m \rightarrow r2$ is of length 8). The replica $r2$ need not be checked for relocation because it is already as close to the root as possible and therefore, it cannot be moved any closer to node l.

We need to move replicas towards the victim node along the path from victim to the sink. It is easy to see that if a replica is placed at the victim node, then cost constraint for all orphans can be satisfied. However, it may not be possible to relocate replicas so that a replica can be placed at the victim. Thus, the victim node must provide the distance, *greatest*, such that if a replica is placed at distance *greatest* (or less) from victim along the path to the root, the cost constraints for all orphan nodes would be satisfied. $greatest = 0$ implies that a replica must be placed at the victim node. Let *max* be the maximum orphan distance received by the victim while scanning for orphan nodes. Then $greatest = Cost - max$.

There are two possible ways in which relocation can be carried out. The first approach involves placing a replica at the victim node, and then try to move replicas (using protocol *Move* discussed in the next section) to check whether a replica can be eliminated. If the replica introduced was unnecessary, then the movement will result in a replica moving to a node which already contains a replica. In this case, one of the replicas can be eliminated.

The other possible approach is to compute whether replicas can be relocated so that a replica can be placed within distance *greatest* from victim. For any neighbor, $Cost\text{--}depth(root, j)$ denotes the distance by which the replica at the *root* can move away from j and still satisfy the cost constraints for any node x reachable via j such that $root = min_rep_x$. Let k be the first node in the path from the root to *victim*. Then, $d = min(Cost\text{--}depth(root, j) : j \in Connect_{root} - \{k\})$ is the distance by which replica at *root* can relocate in the direction of k, where $k \in Connect_{root}$. If $d < w(root, k)$ then the replica cannot move to k. Otherwise, it can at least move to k. Let g be the farthest node along k such that the length of the path from g to *root* is less than or equal to d. Then, the replica at root can move to g but not beyond that. Let the next replica between g and *victim* be at node e. If a replica at the root were to be moved to g then some of the nodes which are connected to e might find g nearer. As a result, it might be possible for e to move towards the *victim* node. Then, $d' = min(Cost - depth(j, e) : j \in Connect_e - \{k'\})$ is the distance by which replica at e can move towards the victim, where k' is the first node in the path from e to *victim*. Note that as a result of moving replica to g, the depth information at e along g will change and has to be recomputed. We continue in this fashion to find by how much replicas can relocate towards the victim node. If a relocation is possible so that a replica can

be placed within distance *greatest* from victim then relocation is performed. The computation for relocation discussed above can be implemented in several ways. One possible way is to accumulate all the information at the *root* so that it can locally compute the distance by which each replica in the path from *root* to *victim* can move. Another possible implementation is to distribute the computation at various nodes in the path from the root to the victim. Both methods are discussed in [2].

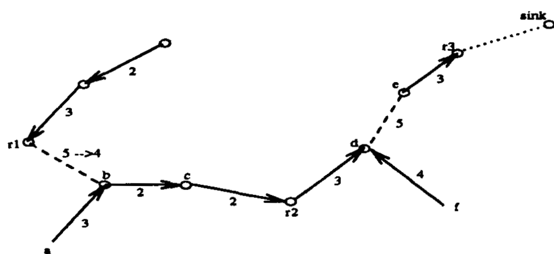

Figure 7: Decrease in an edge cost

5 Decrease in an edge cost

We will now consider the case in which an edge cost decreases. A decrease in an edge cost does not invalidate the cost constraint at any node. However, it might enable us to recompute paths so that some replicas can be removed. One approach is to design *Decrease_Cost* so that whenever a link cost decreases, the protocol is invoked and it ensures that the number of replicas is minimized. We find that this approach will generate a lot of reconfiguration activity (especially if link cost changes are frequent). We have employed an alternative approach in which we have partitioned *Decrease_Cost* into three component protocols. These component protocols may be invoked periodically or whenever link cost decreases. We ensure that these component protocols lead the system to a state with a minimum number of replicas (*i.e.*, if no changes occur for a sufficient period of time and these component protocols are invoked then the number of replicas is minimized).

Protocol *Decrease_Cost* tries to ensure that replicas are placed as close to the sink as possible (and if there is a replica at the sink node then we try to merge it with a neighboring replica). The three component protocols are *Decrease_Response*, *Change_Depth* and *Move*. Assume that the cost of (u, v) has changed, where $parent_u = v$. In this case, u sends a message to all replicas in Rep_set_u informing them of the change. Protocol *Decrease_Response* is initiated at a replica site r when it learns of the change (or on a periodic basis). The purpose of this protocol is to ensure that if there exists a node in the path from r to the sink which is at distance less than or equal to *Cost* then it should be connected to r. For example, in Figure 7, let the cost of $(r1, b)$ decrease from 5 to 4. Then, *Decrease_Response* will get initiated by $r1$. This protocol will determine that both b and c are at distance ≤ 7 and can connect to $r1$ (assuming $Cost = 7$). This recomputation of paths will change the depth information. To update this information, protocol *Change_Depth* is invoked. For example, in Figure 7, $depth(r2, c)$ is updated to 0 (since $r2$ is no longer min_rep_c). This allows the movement of $r2$ towards the sink (note that earlier $r2$ could not have moved). This movement is accomplished by invoking *Move* at $r2$. Assume that $r2$ has moved the required distance (in this case, it can move to d). Due to this movement of a replica, there might exist a node j in the path from the replica to the sink such that j is now within *Cost* distance from this replica. For example, in Figure 7, $r2$ can move to d. As a result, e can connect to $r2$. To recompute these paths, protocol *Decrease_Response* is initiated. This invocation might lead to the movement of the next replica in the path to the sink. For example, in Figure 7, e will connect to $r2$. This will result in updating $depth(r3, e)$ to 0, which will enable $r3$ to move towards the sink. If a replica moves to a site which already contains a replica then one of the replicas is eliminated. If there is a replica at the sink, we try to move it to check if it can be merged with a neighboring replica.

In the following subsections, we will discuss the three component protocols.

5.1 Protocol *Decrease_Response*

We will discuss the invocation of this protocol by a replica r. To initiate this protocol, r sends a message DECREASE$(0, r)$, where the first argument denote the distance to the nearest replica and the second argument is the replica identity. In general, let j receive a message DECREASE(l, r) from i. If $l + w(i, j) < Cost$ then j can connect to min_rep_i (if it is not already connected to it). This might provide some flexibility for min_rep_j to move towards

the sink (which is the next replica in the path from j to the sink). Thus, node j behaves as follows: if $l + w(i,j) < Cost$ then j updates $pref_j = i$, $min_rep_j = r$, $dist_rep_j = l + w(i,j)$. In addition, it sends DECREASE($dist_rep_j, min_rep_j$) to $parent_j$. If $l + w(i,j) > Cost$ then j cannot connect to r. It sends an DEC_CK message to i. Further, as a result of these recomputation, the depth information of the nodes in the path from j to min_rep_j must have changed. To update this information, it initiates *Change_depth*. On receiving DEC_ACK message, i also initiates *Change_depth*. For example, in Figure 7, $r1$ will send a DECREASE($0, r1$) message to b. Hence, b will connect to $r1$ and send DECREASE($4, r1$) to c. On receiving this message, c will also connect to $r1$.

5.2 Protocol *Change_depth*

We will discuss the initiation of this protocol by a node i. To initiate this protocol, node i does the following. It first updates max_depth_i. It then sends a message CHANGE(max_depth_i) to $pref_i$. When a node j receives CHANGE(l) message from k, it updates $depth(j,k)$ to $l + w(j,k)$. If this results in a change in max_depth_j then it sends CHANGE(max_depth_j) to $pref_j$. When a replica node r receives this message, it behaves as follows. If $max(depth(r,j) : j \neq parent_r)$ is less than $Cost$ then r can potentially move towards the sink. Therefore, it initiates protocol *Move*. For example, when $r2$ receives the DECREASE message from c, it will send N_ACK message to c and initiate *Change_Depth*. Hence, it will update $depth(r2, d)$ to 0 and initiate *Move*.

5.3 Protocol *Move*

The purpose of this protocol is to move the replica at the initiator site r towards the sink. Let $x = max(depth(r,j) : j \neq parent_r)$. Then, r can move distance $Cost - x$ towards the sink (which is in the direction of $parent_r$) without violating the cost for any node. For example, in Figure 7, $r2$ can move distance 5 in the direction of the sink. In this case, r moves in the direction of the sink as much as possible, updating the depth information and parent pointers as it moves. If j was the first node in the path from r to the sink and $w(r,j) < x$ then r can move to j. In this case, the following operations are performed: $parent_r = j$, $depth(j,r) = max_depth_r + w(r,j)$. After performing these operations, the replica is at site j. At site j, it again tries to move one hop towards the sink. This continues until the replica cannot move any closer to the sink. After moving the required distance, a message is sent to all nodes connected to this replica to change their distance to this replica. For example, in Figure 7, $r2$ can move only up to d (since $Cost = 7$). Assume that the replica has moved to node k. Let the next replica in path from k to the sink be at m. Due to this movement, some nodes which were earlier connected to m might be less than $Cost$ distance from the replica at k. For example, in Figure 7, e can connect to $r2$ if it is moved to node d. To do this recomputation, the replica initiates *Decrease_Response*.

6 Conclusion

We studied the problem where several replicas are placed in a network and each site has to obtain a path to its nearest replica site, such that the length of the path is less than or equal to $Cost$. The first protocol, *Path_To_Nearest*, in our system establishes a path to the nearest replica for each site. The protocol *Increase_Cost* rconfigures the system so that the cost constraints are maintained for each node in the presence of link cost increases. The protocol *Decrease_Cost* reduces the number of replicas to bound the cost of write operations. We employed discrete event simulation to validate and obtain performance results. Due to space constraints, the results are given in [2].

References

[1] Bernstein, A.J. and Wuu, G.T. In Proc. of ACM Sym. on Principles of Distributed Computing, 1984.

[2] Bommareddy, M. Replica Placement in a dynamic network, *M.S. thesis*, Kansas State University, 1993

[3] Dijkstra, E. and Scholten, C. Termination Detection for diffusng computations, Info. Processing Letters, 11(1), 1980.

[4] Garey, M. and Johnson, D. Computers and Intractability, 1979.

[5] Liskov, B. Highly available distributed services, *Prog. Methodology Group Memo 52*, MIT, 1987.

[6] Merlin, P. and Segall, A. A failsafe distributed routing protocol. *IEEE Trans. on Comm.*, 27(9), 1979.

[7] Wolfson, O. and Jajodia, S. Dynamic maintenance of replicated data, Proc. of ACM Sym. on Principles of Database systems.