Propagating Updates in a Highly Replicated Database

Tony P. Ng

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

In this paper we consider the problem of propagating messages to a large number of hosts in a distributed database. The work of propagating the message is distributed and minimized among the hosts by arranging them into a minimum-cost spanning tree. Only the cooperation of the hosts that belong to the database is required. Transient and permanent failures are handled in our protocol. Eventual delivery and propagation within a network partition are guaranteed despite failures. This is achieved without requiring a large amount of information to be stored and maintained by each host, even when many hosts in the database may initiate updates.

1 Introduction

In this paper we consider the problem of propagating updates in a highly replicated database. Such a database is partially or completely replicated in a large number of hosts (e.g., more than 1,000 replicas) connected by a communication network. Possible applications include some form of yellow page (name) servers, routing tables, library catalogs, phone directories, stock market databases, etc. Replication provides ease of access, availability, and reliability in these applications.

In the applications described above, updates to the database may be initiated by a few or by all of the hosts that contain a replica. In order to maintain consistency of the replicas, these updates have to be propagated to all replicas eventually. In this paper, we will consider the more general problem and assume that updates may be initiated by any host that contains a replica. In section 7 we will argue that a solution that allows only one host to initiate updates may not be easily extensible to one that allows multiple hosts.

Typically, an update may affect only a small portion of the database. Although batching the updates is a possibility (as is done for paper phone books), the need for up-to-date information, decentralized control, and efficient use of network resources may necessitate propagating the updates individually.

Propagating updates in such an environment requires careful programming. First, due to the large number of replicas, our protocol should distribute any computation uniformly among the participating hosts. For example, sending an update from the source repeatedly to each other replica may have a prohibitively high cost. The same message may travel over the same communication link repeatedly and tie up the resources at the source for a very long period of time. Similarly, requiring each host to keep information about every other host is probably a bad idea.

2 System Model

2.1 Hosts, Imps, and Communication Network

We assume that the hosts in our system are connected by a point-to-point communication network. Hosts are connected to the network through Imps, which are connected to one another with communication links. Our communication network GC can be defined as:

\[ GC = (Nc, Ec), \]

where \( Nc = N_i \cup N_j \), \( N_i \) is the set of Imps, \( N_j \) is the set of hosts, and \( Ec \subset Nc \times Nc \) is the set of connection links. In other words, \( (n_1, n_2) \in Ec \) if there is a communication link between the Imps/Hosts \( n_1 \) and \( n_2 \).

We will assume that each host in our system contains a complete replica of the database. We will refer to the set of Imps and hosts as the nodes in our system. A node can serve as an Imp and a host simultaneously. It should be noted that our network may contain many other hosts that do not run our database application. We will ignore such nodes unless they serve as Imps also. The presence of these hosts makes broadcast protocols that broadcast to every host through every Imp [AE84, DM78, RS86, SA83] inefficient.

Nodes and communication links can fail. The failure can be transient, or it can be permanent as long as the rest of the system remains connected. Allowing permanent failures simplifies system reconfiguration. Nodes and links can be removed without having to synchronize with the propagation of updates. The removed component can be regarded as having failed permanently.

CH2840-7/90/0000/0529$01.00 © 1990 IEEE
Furthermore, sometimes a "transient" failure can last for days. By expecting permanent failures, our protocol avoids waiting for such transient failures to end.

Messages sent by our network may be lost, duplicated, or reordered. We assume that messages are not garbled.

2.2 Database Updates

In our protocol an update message may be delivered to the same host more than once. We believe that handling duplicate messages is best left to the database application itself. In general, eliminating duplicate messages requires remembering the receipt of a message until it is certain that no more duplicates will be received. Such information would have to be remembered for a long time in our environment, given the relatively long time to propagate an update message to every host. It would also require our protocol implementation to perform checking for each message received. Furthermore, if such checking requires locally stored information, stable storage (which is the only kind of storage that survives node failure) has to be used. On the other hand, a database application may be able to handle duplicate messages more easily, such as by making the processing of an update message idempotent. For example, Grapevine[BLNS82] associates a timestamp with update message idempotent. For example, Grapevine[BLNS82]

Our protocol also does not guarantee that messages are delivered in the order they are sent. If needed, ordered delivery can be implemented easily with sequence numbers.

3 Basic Solution

The basic idea of our protocol is to construct a minimum-cost spanning tree \( T = (N_T, E_T) \) connecting all the hosts (which contain replicas), where \( E_T \subset N_T \times N_T \). Various metrics, such as delivery time or percentage of bandwidth consumed, can be used to measure costs. An update message initiated by any host is propagated along the edges of this spanning tree. For example, for the network in figure 1(a), its minimum-cost spanning tree is represented in figure 1(b). Note that the edges in the minimum-cost spanning tree connect only hosts and never imp. Although a spanning tree that allows edges connecting to imp may be less expensive, our protocol involves only hosts and does not require programming on an imp, which, as argued in [GK88b, GK88a], can be difficult in a large network. In any case, our protocols can be easily extended if it can be run on an imp.

In this paper, we will ignore short-term changes of the cost of sending a message over a communication path. Although the changes may make the optimal spanning tree sub-optimal, we believe that the change would only represent a small percentage of the total cost of propagation. As a result, we avoid the complexity and overhead of dynamically responding to such short-term changes. The minimum-cost spanning tree can be reconstructed infrequently to take care of any long-term changes.

Before describing our protocol further, we will define the following terminology assuming that a message \( m \) is originated at a host \( h \):

- the logical neighbors of \( h \) is \( \{a : (h, a) \in E_T\} \). For example, the host \( g \) in figure 1(b) is the logical neighbor of \( f \) and \( d \).
- the physical neighbors of \( h \) is \( \{a : (h, a) \in E_T\} \). For example, the imp \( i \) in figure 1(a) is a physical neighbor of \( f \).

Figure 1: A Communication Network and Its Minimum-Cost Spanning Tree

- children\(_h\): the children of \( h \) are the logical neighbors of \( h \) such that \( h \) is the path in \( T \) from those neighbors to \( h \). In figure 1(b), children\(_h\)(a) = \{b, g\}. If \( h = o \), then the children and the logical neighbors are identical.
- parent\(_h\): the parent of \( h \) is the logical neighbor of \( h \) such that the parent is on the path in \( T \) from \( h \) to \( o \). In figure 1(b), parent\(_h\)(c) = o. It is undefined if \( h = a \).
- descendants\(_h\): the descendants of \( h \) are \( h \) itself and the transitive closure of children\(_h\). In the example in figure 1(b), descendants\(_h\)(a) = \{a, b, f, g\}. Note that for any \( d \) \in descendants\(_h\)\(, descendants_ah = descendants_h\). For example, descendants\(_h\)(a) = descendants\(_h\)\(a\).
- the logical distance between \( h \) and \( j \) is the number of edges in the path between \( h \) and \( j \) in \( T \). For example, the logical distance between \( a \) and \( i \) in figure 1(b) is 2.

In the rest of this paper, "neighbors" will mean "logical neighbors" unless specified otherwise.

In our protocol, each host \( h \) would have to remember only its neighbors. When a message that originates at \( c \) is received, it is propagated to all the children (children\(_c\)). If no failures occur during the propagation of an update message, it will be sent along the edges of \( T \) to all the hosts eventually. In other words, through each child \( c \), the message would be eventually propagated to descendants\(_c\) (which is equal to descendants\(_c\)).

Two types of failures may occur during an update message propagation. First, in trying to deliver a message to \( c \), \( h \) may discover that \( c \) is unreachable. Waiting for \( c \) to recover may cause excessive delay in delivering the message to the descendants of \( c \). Worse, \( c \) may have failed permanently. Second, \( c \) may have received the message from \( h \) but failed before propagating the message to all its descendants.
how the recovery procedure is programmed. While its ble, we would like to avoid techniques like flooding[BBR64] which propagate the update message to as many descendants invoked. The goal of the recovery procedure is to propagate the return an communication network for each system configuration. If new hosts are added, we can use removed permanently, the minimum-cost spanning tree has to be reconstructed if the removed component is used in at least one of the shortest paths in the minimum-cost spanning tree. In order to minimize the cost of recomputing minimum-cost spanning trees, such recomputation can be performed infrequently while a slightly sub-optimal tree is used in the interim.

3.2 Recovery Procedure

When unreachability of a child c is detected by a host h before c returns an ack message, h can conceivably send the update message to each of c's descendants. However, since c may have many descendants, doing so may overwhelm the resources of h and create the same problem that we intend to solve originally. In our protocol, h would attempt to send the update message to a small backup set of hosts backup(h, c) \subseteq descendants(c). Each host in backup(h, c) will be tried successively until the update message is successfully delivered to one of them. Again, h will wait for an ack message from the descendant that received the update message. If all alternatives fail, h can pause momentarily and retry later, unless all hosts in (c) \cup backup(h, c) have failed permanently1. Although it is possible that there are some

1We assume that our failure detection facility can distinguish between transient and permanent failures.

hosts in descendants(h, c) \rightarrow (c) \rightarrow backup(h, c) that have not failed permanently, we ignore this possibility given that permanent failures are rare and the backup set is not extremely small. How the membership of backup, c is determined is described in section 3.3. Refinements to that procedure is developed in sections 5 and 6.

At the backup host d that receives the update message, it will propagate the message as if the message were originated at d and needed to be propagated in the sub-tree descendants(c). In order to delimit the smaller scope of this message, a scope delimiter component is appended to the message. In the rest of this paper, we will use the word "scope" to refer to the set of hosts to which a message should be delivered (the hosts in descendants(c) in this case), and "scope delimiter" to refer to the representation we use to identify the scope. In this case the scope delimiter consists of the single identifier h, since descendants(c) is uniquely defined by h. The scope delimiter can be used by the recipient of a message to determine whether a child is within the scope indicated (see figures 3 and 4). Thus for example when c eventually receives the message, it will not try to deliver the message to h. When d receives ack messages from all its children2 (the update message has been propagated to all the hosts in the sub-tree), it can return an ack message to h. Conceptually, we can imagine that the link between h and c in the minimum-cost spanning tree has been severed temporarily and a link between h and d is added.

If another failure of reaching a child c is encountered by a host h' during the propagation of the update within descendants(c), the recovery procedure can be invoked again. The responsibility of delivering to descendants(h', c) \cap descendants(c) can be delegated to one of backup(h', c). Again, the scope of the even smaller subtree can be uniquely defined by appending the identifiers h and h'. Since failures are expected to be rare, the length of the scope delimiter would usually be very short. An example of how an update message may propagate appears in figure 2, in which we assume that a message m is originated at host f and host a has failed. We assume a network topology as the one in figure 1(a). The numbers indicate the order in which messages are delivered. We assume that b is the backup when g cannot deliver to a and c is the backup when b cannot deliver to a. When c cannot deliver to a it will keep trying periodically because a is the only member in the subtree descendants(a) that are not beyond g or b. If c failed before delivering to a, the failure will be detected by b and c will resume trying to deliver to a. Similarly b's failure will be covered by g. Note that in this example, all hosts receive the message without being blocked by the failure of a, even though a is an interior node in our propagating spanning tree.

Since the message is being regarded as having been originated at d, it is sent to all of d's neighbors.

---

Figure 2: An Example of Update Message Propagation
while true do
  receive message \((m, a)\) from \(p\) /
  s is scope delimiter */
  for each \(c\) in \((h's\ neighbors - \{p\})\) /*i.e., children of \(h\) */
  if inside\((c, s)\) then create background process to
  execute deliver\((m, s, c)\)
wait for all background processes to finish
send ack message to \(p\)
end while

deliver\((m, s, c)\)
  send \((m, a)\) to \(c\) wait for reply from \(c\)
  case 1 - ack message from \(c\) received: return
  case 2 - unreachable of \(c\) detected:
    for each \(d\) in backup\((h, c)\)
      if inside\((d, s)\) then do
        send \((m, s)|h\) to \(d\) and wait for reply
        case 1 - ack message from \(d\) received: return
        case 2 - unreachable of \(d\) detected: continue
      end if
    end case
  pause and start from beginning unless \(c\) and all hosts
  in backup\((h, c)\) have failed permanently
end deliver

Figure 3: Propagation Protocol Implementation for a Host \(h\)

Summarizing, our update message propagation protocol executed by a host \(h\) can be represented as in figure 3. The function inside returns whether the host indicated by the first parameter is inside the sub-tree indicated by the second argument (scope delimiter). The scope delimiter is a list of host identifiers, which are hosts that are not in the sub-tree but neighbors to the leaves of the sub-tree. In other words, the sub-tree is represented with a list implicitly. An implementation of inside is given in figure 4. In this figure, inside is implemented by testing whether the host \(y\) is beyond the boundary hosts (not between the boundary hosts and the local host). The function between can be implemented without knowing the global topology of the minimum-cost spanning tree. We do so by naming all the hosts hierarchically. First, an arbitrary host can be picked to be the root of the naming hierarchy. Then all hosts are named with the convention that their names must have their parent's name as a prefix. With the naming hierarchy, between can be implemented as easily as in figure 4.

In the program fragment in figure 4, we have not dealt with the possibility that an ack message from \(h\) to \(p\) may be lost. To avoid waiting forever, \(p\) can probe the progress of \(h\) periodically (in addition to detecting \(h's unreachable\)). If \(h\) has no memory of the message \(m\), it may be because the initial message to \(h\) is lost due to some failure, or because the ack message from \(h\) is lost and \(h\) has deleted all memory of \(m\). Unable to distinguish between the two, \(p\) can resend \(m\). If the latter is true, the resending would create largely harmless duplicate messages. This can be made unlikely to happen by using different techniques, e.g., sending the ack message multiple times, or remembering the ack message for at least the probe interval.

inside\((a, s)\): boolean
  if \(s = "-"\) then return \(-\)between\((\text{myself}, \text{message sender}, a)\)
  return \(-\)between\((\text{myself}, \text{first}(s), a)\) and inside\((a, \text{rest}(s))\)
  /* first\((s)\) returns the first identifier in \(s\) */
  /* rest\((s)\) returns the rest */
end inside

between\((a, b, c)\): boolean
  common := common prefix of \(a\) and \(c\)
  if common is a prefix of \(b\) and \(b\) is a prefix of \(a\) or \(c\)
    then return true
  else return false
end between

Figure 4: Implementation of Inside and Between

backup := \(\emptyset\); temp := physical neighbors of \(h\); explored := \(\{h\}\)
while temp \(\neq \emptyset\) do
  temp := temp - \(\{x\}\) /* \(x\) chosen randomly from temp */
  explored := explored \(\cup\) \(\{x\}\)
  if \(x\) is descendant\((h)\)
    then backup := backup \(\cup\) \(\{x\}\)
  else temp := temp \(\cup\) physical neighbors of \(x\) - explored
end while
backup\((h, c)\) := backup

Figure 5: Determining the Membership of backup\((h, c)\)

3.3 Determining the Membership of Backups

Suppose \(h\) received a message with the responsibility to deliver it to all the hosts in the scope \(s\) and one of its children \(c\) is unreachable. In determining the membership of backup\((h, c)\), we would like to maintain the property that if there exists a physical path from \(h\) to one or more hosts in \(s\), then at least one of those hosts is in backup\((h, c)\). We will show that this property will lead to another property that two hosts in the same network partition are guaranteed to receive the same set of messages under certain conditions. We will describe one way of determining the backup set in this section. This computation can be performed offline for each pair of \(h\) and \(c\). We will describe a way to reduce the size of the backup set in section 5 by guaranteeing the property described above with only a high probability. We will assume the \(s = \text{descendants}\((h)\) and address the case when \(s \subset \text{descendants}\((h)\) in section 6.

In the worst case, we can have backup\((h, c) = \text{descendants}\((c)\). To reduce the size of backup\((h, c)\), we can take advantage of the fact that some hosts serve as imps also. Consequently, the unreachable of a host may imply the unreachable of other hosts. Figure 5 shows one way of determining the backup set. The procedure in figure 5 starts with \(h\) and follows all distinct paths that lead from \(h\) to some host in \(\text{descendants}\((c)\). A path is followed until it reaches the first host in \(\text{descendants}\((c)\) (i.e., some host in \(\text{descendants}\((c)\) may not be explored because all paths from \(h\) to it have to pass through some other host(s) in \(\text{descendants}\((c)\) first). The subset of the host in \(\text{descendants}\((c)\) that are explored in figure 5 is used as backup\((h, c)\).
4 Properties of the Basic Solution

In this section we will argue that given our propagating protocol and our algorithm of determining backup(h, c), every host in N_f that has not failed permanently will receive (at least) one copy of every message eventually. In addition, we will argue that the protocol terminates and two hosts in the same partition will receive the same set of messages under certain conditions.

4.1 Eventual Delivery

Consider a host h that has received a message (m, s) to be responsible for transmitting the message to every host in the scope indicated by the scope delimiter s. The responsibility to one of these hosts d is removed only when d actually receives a copy of the message. Given our protocol, there is a chain of hosts responsible for the delivery to d for any d that has not received a copy. The chain starts with the host o that originated the message, which has a scope that encompasses the entire spanning tree T. Since a host is eventually connected to all other hosts that have not failed, the chain grows eventually when the message propagates to the child between o and d or a backup for that child. As explained in section 3.2, we will ignore the possibility that the child and all the backups have failed permanently. The chain can shrink when some of the hosts in the chain fail. The chain grows again when the failure is detected and the responsibility of delivering to d is given to other hosts.

If the chain never shrinks to nil permanently, at least one host will be responsible to deliver the message to d. Although it is possible that the chain can grow and shrink indefinitely but never delivering to d, its probability approaches zero. If d has not failed permanently, a copy of the message will be delivered to it after the chain has grown a finite number of times. The chain grows only a finite number of times because each time the scope becomes strictly smaller.

In order to guard against the chain from shrinking to nil permanently, the originator o of the message can use a two-phase commit protocol [Gra78] to deliver the message to its children. Only when a child is assured that all other children have a copy in their stable storage [Lam80] would it start propagating. The two-phase commit protocol guarantees that any chain would have at least two hosts. Since it is highly unlikely to have more than one host fail permanently in the time to propagate the message, we can ignore this possibility. If this is not true, the grandchildren of o can be included in the two-phase commit protocol.

4.2 Termination

Our protocol terminates in a host h when it has propagated a message to all its children, or some backup for a child that it cannot reach. h can also stop trying to deliver to a child c and its backups when they have failed permanently. As described in section 3.2, we will assume that all hosts in descendants(c) must have failed permanently in that case.

With the scope of a message decreasing strictly each time it is propagated, a message would only be propagated a finite number of times.

4.3 Hosts in the Same Partition Receive the Same Messages

Suppose hosts x and y are in the same network partition and x has received a message (m, z) but y has not. In this section we will argue that y will receive m if z is an ancestor of y. We define an ancestor of y as a host such that the scope of responsibility it receives includes y.

We will show by induction on the size of the scope s that y will receive the message m. In the base case of our induction, the size of the scope s is 2. In that case y has to be the neighbor of z. Since they are in the same partition, y will receive m from z when z tries to propagate the message. Consider the general case in which the size is larger than 2 and y is not a neighbor of z. Consider the child c of z such that c is between z and y. If z can communicate with c, then c will receive the message m. Otherwise, since there is a path of physical links connecting z and y, at least one host in backup(z, c) will be in this partition. This is because the procedure in figure 6 chooses either y or some other host on a path to y as a backup. In any case, either c or the backup will receive m with a scope of responsibility that includes y but strictly smaller than s. Using induction, that host and y will have the same set of messages, which implies that y will receive m.

Two hosts x and y in the same partition will not receive the same set of messages only when neither host is an ancestor of the other. Moreover, we should note that z and y share at least one common ancestor because the origin of a message has a scope that includes all hosts. Also, each host must have received its messages directly or indirectly from its ancestors. Consequently, if z has received a message m but y has not, then it could only have happened because a network partition occurred between a common ancestor a and z (and between a and y) after a has delivered m to z or to a child between z and itself.

5 Determining the Membership of Backup Probabilistically

In this section we will consider an improvement to our basic protocol. Previously in section 3.3, we have determined the membership of the backups of a child c using a breadth-first search. All descendants of c that can be reached without passing through another descendant is included in the backup set. This procedure has the problem that if hosts are in general located at the fringe of our communication network, practically all descendants of c will be included. This defeats the purpose of our protocol, which is to avoid one host sending to many other hosts.

In this section we consider an alternative way of determining the backup set of a child. We use a probabilistic approach. If all of the descendants in the backup set are unreachable, then all other descendants should be unreachable with a high probability p (instead of with absolute guarantee).

Assuming that all nodes and communication links fail independently, one possible way for a host h to determine such a backup set for a child c is described in figure 6. Step (3) of the algorithm in figure 6 can be implemented using the methods described in [Ros77]. In [Ros77], a communication network is broken into smaller subnetworks. Each subnetwork has boundary nodes that are shared by other subnetworks. States representing whether each node/link in a subnetwork is functional can be classified into equivalence classes (the equivalence relation depends on the probability to be calculated). The number of equivalence classes is very sensitive to the number of boundary hosts. Subnetworks are combined to become larger subnetworks. The probability of an equivalence class of the larger subnetwork is calculated by considering the cartesian product of the equivalence classes of
In any case, much of the computation in an iteration of the while loop can be re-used because adding an extra element in the backup set would not change the equivalence classes of most of the sub-networks. We expect the number of iterations required for the while loop to be small.

6 Recovery for Smaller Scopes

Sections 3.3 and 5 assume that the scope $s$ of a message sent by a host $h$ to a child $c$ is $\text{descendants}_s(c)$. This is not true if failures have been encountered before the message is delivered to $h$. It is possible that the smaller scope $s$ has no intersection with the backup sets determined in sections 3.3 and 5. Recall that in figure 3 a backup host is used only if it is in $s$. This creates two problems. First, the child $c$ may have failed permanently, and since there are no backups (in the scope $s$), $h$ is unable to deliver the message to the hosts in the scope $s$. Second, there may be a path of physical links connecting $h$ with some host in $s$, but since there are no backups, that host will not receive the message until after $c$ has recovered, despite the existence of a functional path. Ideally, the procedures in figures 5 and 6 can be repeated for each possible scope and each pair of $h$ and $c$. However, since the number of possible scopes can be prohibitively large, this is not a feasible solution.

In order to solve the first problem, we include in $\text{backup}(h, c)$ all hosts that are within a logical distance $k$ from $h$ and are in the set $\text{descendants}_s(c)$, where $k$ is small constant. Since the hosts in any scope $s$ must form a sub-tree that includes $c$, the size of the set $\text{backup}(h, c) \cap s$ must be at least $k$ if the size of $s$ is at least $k$, and the set will include $s$ otherwise. The value of $k$ can be chosen large enough such that it is highly unlikely for $k$ hosts to have failed permanently. If the size of $s$ is less than $k$, then $s \subseteq \text{backup}(h, c)$ and the recovery procedure will only give up after all the hosts in $s$ have failed permanently.

In order to solve the second problem, we define the transitive closure of $\text{backup}(h, c)$ to be:

$$\text{backup}(h, c)^* = \text{backup}(h, c) \cup \left\{ c \right\} \cup \bigcup_{d \in \text{backup}(h, c)^{\circ}} \text{backup}(h, \text{child}(d))$$

where $\text{child}(h)$ is the child of $h$ that is on the path from $h$ to $c^*$. We modify the recovery procedure so that if the set $\text{backup}(h, c)^{\circ} \cap s$ is empty or if none of the backup hosts in that set is reachable, the set $\text{backup}(h, c)^* \cap s$ will be used as the backups.

Using $\text{backup}(h, c)^*$ as the backup set preserves the properties described in sections 3.3 and 5. Suppose there is a path of physical links from $h$ to a host $d$ in $s$ and suppose we use the procedure in figure 5 to determine $\text{backup}(h, c)$. Either $d$ is in $\text{backup}(h, c)$ or it is not. If it is not, then some host $b$ on that path must be in $\text{backup}(h, c)$. By induction on the length of that path, $d$ must be a member of the set $\text{backup}(h, \text{child}(d))^*$. In both cases, $d$ is a member of $\text{backup}(h, c)^*$. Thus $h$ will be able to deliver to at least one host in $\text{backup}(h, c)^*$ if there is a path from $h$ to any host in $s$.

If $\text{backup}(h, c)$ is determined probabilistically as in figure 6, we can follow a similar argument and show that $d$ will be in $\text{backup}(h, c)^*$ with a high probability.

The membership of $\text{backup}(h, c)^* \cap s$ should be determined dynamically after the possibilities in $\text{backup}(h, c)^* \cap s$ are exhausted.

\footnote{In other words, in a "bottom-up" order. The bottom-up order in general minimizes the number of boundary hosts in networks with little or localised redundancy. Heuristic methods are described in [BBD72] to determine an efficient order of combination.}
since $s$ can take on many different values. When needed, $h$ can propagate a query message containing the scope delimiter of $s$ to each of its reachable backups $b$ in $\text{backup}(h,c)$, which would return the membership of $\text{backup}(h,\text{child}(b))$. $h$ can then try to send the update message to these second-level backups that are in $s$. If none of the second-level backups in $s$ are reachable, $h$ can send query messages to the second-level backups to determine the third-level backups, etc. The number of levels that $h$ has to try is bound by the number of hosts that are on a physical path from $h$ to a reachable host in $s$. We expect the actual number to be much smaller than this bound.

7 Related Work

[DM78, SA83] describe broadcast protocols that make use of spanning trees. The latter guarantees that messages are delivered to every host, without duplication, and in the order sent. The former does not. The protocols are closely coupled with the routing algorithm in the communication network in which the spanning tree is determined by the performance measure used by the routing algorithm. Such coupling implies that broadcast has to be supported by the subnet, and cannot be added by the application. The protocols described in [DM78, SA83] broadcast a message to all the hosts connected by a communication network. However, we believe that most applications really require multicast, in which only a subset of the hosts have to be reached. The size of the subset is typically a small percentage of the total number of hosts in the network, even though the number of hosts in the subset is large enough that directly sending to each host is too expensive. The protocol in [SA83] requires each host to store all the messages that may be sent by every other host between two routing table updates. This is unacceptable for our applications where a large number of hosts may initiate updates.

[Pv80] also describes the use of broadcast trees to propagate messages to a group of hosts. The authors analyze the delay of their protocols when a host can send one message to another host at a time. However, since their protocols are used in a simulation environment, the authors did not specify how the broadcast trees are constructed or any procedures that can be used to deal with failures.

[Ros86] describes a flooding protocol for routing updates in the Arpanet. Flooding has the advantages of speed and that if two hosts are connected by a physical path, flooding guarantees that a message received by one host will be received by the other. We do not consider flooding because of the large number of messages it generates and the difficulty of keeping track of when a message is too "old" to be propagated. In addition, since we are interested of broadcasting to only a subset of hosts in the network, flooding has to be applied along logical edges connecting this subset of hosts. As such, we cannot guarantee that two hosts connected by a physical path receive the same messages anymore.

[AE84] describes a protocol that propagates messages along a knowledge tree. The tree is constructed by having each node establish a link with the neighbor that has received the largest number of broadcast messages. It is not clear from the paper how the protocol deal with failures and how much overhead is incurred in constructing the knowledge tree. Finally, the protocol works only when a message is to be delivered to all the nodes in the network, rather than only a subset of the hosts.

[GR88b] describes a protocol in which hosts are divided into clusters with cluster leaders. The members in a cluster receive broadcast messages from their leader, and the cluster leaders are organized into a tree similar to the knowledge tree in [AE84]. The paper describes how the cluster tree can be rearranged dynamically. [GR88b] is similar to our protocol in that it does not require the programming of imps and can be run on only a subset of the hosts in the system. One of the disadvantages of their protocol is that each host has to keep track of large amount of information. For each possible origin of broadcast messages, a host has to keep track of which messages it thinks other hosts have received. Assuming the amount of information to encode which messages have been received by a host is on the average $i$, and there are $n$ hosts in the system, each of which can be a source of broadcast message, the total amount of information maintained by each host is roughly $i^2$. The amount of space or the work required to maintain this information may become unacceptably large for large $n$.

[GR88a] describes a protocol that combines the use of an unreliable multicast facility (provided by the network) and a redelivery algorithm in case of failures. The initiation of the redelivery algorithm requires some form of failure detection, which in turn depends on the periodic transmission of (null) messages. Upon detection of a failure, a host $h$ will try to contact element of a subset of other hosts successively. The order in which they are contacted are determined by a priority list stored locally. By guaranteeing certain conditions of these priority lists, the authors are able to show that all hosts in a network partition will configure themselves into a tree as they contact one another. The priorities (joint completeness and acyclicity) impose a total order on all the hosts. On the average, the size of each host’s priority list is of the order $O(n)$ where $n$ is the total number of hosts. The idea of using a priority list is similar to our idea of a backup list. However, we expect the size of our backup lists to be typically much smaller and consequently would not tie up the resources of a host for a long period of time during a failure (redelivery). In addition, a different priority list is used for each possible origin of a message. Assuming that all $n$ hosts may initiate messages, the memory requirement at each host is $O(n^2)$.

Probabilistic algorithms are proposed in [Dv87]. In those algorithms hosts propagate updates to a few other randomly chosen hosts, and, in order to avoid missing some hosts permanently, also periodically compare its data with randomly chosen partners. Since hosts are chosen randomly, there is no upper bound on the delay of an update propagation even when no failures occur. Furthermore, due to the probabilistic nature, the amount of traffic generated are typically several times more than the minimum required.

Several other broadcast algorithms have been proposed [Bir85, LG88, NCN88]. However, these algorithms are geared toward a small set of recipients. Thus they are not suitable for the applications discussed in this paper.

8 Conclusion

In this paper we have presented a protocol to propagate update messages in a highly replicated distributed database. The protocol has the following features:

1. The work of propagating update messages is distributed evenly among the interior hosts of a spanning tree. The size of the spanning tree depends on the application, and is not necessarily equal to the total number of nodes in the
communication network. The latter is assumed by most broadcast protocols.

2. If no failures occur, the cost of propagation is minimum under the constraint that no programming is required on the imps of the communication network.

3. If failures occur, a sufficient condition for two hosts within the same partition to receive the same set of message is that if one is an ancestor of the other.

4. The fact that nodes may be frequently added or removed in a large network is taken into consideration in our algorithms.

5. The protocol delivers messages reliably despite network partitions or permanent node failures.

6. The amount of information kept in each host to run the protocol is small (less than O(n)) even when update messages may be initiated by many different source. The topology of the spanning tree and the membership of the backup sets can be computed offline.

We believe that these features are essential to any practical solution to our problem.

References


