CHAPTER 3

Overlay Protocol

This document contains a descirption of the Hypercube (HC) and Delaunay Triangulation (DT) protocols. The description is from the documentation of Version 2 (2002).

This document also describes an extension to the DT protocol to include Buddylist and Broadcast Rendezvous mechanisms.

http://www.cs.virginia.edu/%7Emngroup/hypercast/documentation.html

An update of this document is pending.

Overview

HyperCast builds overlay networks that have a specified topology. Currently, HyperCast can build two types of overlay network topologies, logical hypercubes and logical Delaunay triangulations. These protocols are described in more detail in [BEAM99] and [LIEBE01b].

The overlay protocol of an overlay socket in HyperCast is implemented by a component which is called overlay node. The overlay node component adds and maintains the membership of an overlay socket in an overlay network.

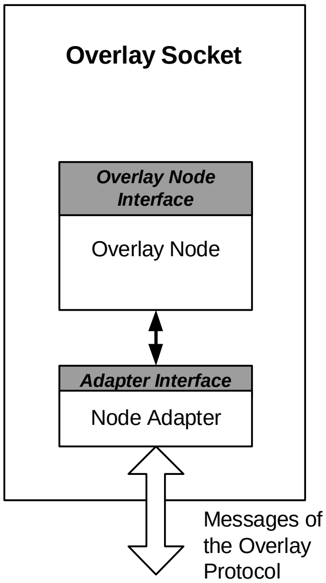


Figure 3.1. The overlay node component of an overlay socket.

Across different overlay protocols can vary widely, the overlay nodes for all overlay protocols share the following properties.

- An overlay node exchanges messages with other overlay nodes in the same overlay network. The overlay node runs a finite state machine which performs actions when timers expire and when messages are received.

- The overlay node communiates with a network adapter which transforms protocol messages int datagrams. The node adapter also maintains the timers of the overlay protocol.
- Overlay nodes distinguish between logical and physcial addresses. A logical address identifies a member in the overlay network, and is derived from some logical address space. Examples of logical addresses are a binary string or coordinates. A physical address is a transport layer endpoint in the network over which the overlay network is run. In HyperCast 2.0, the underlying network is an IP network. Therefore, physical addresses are a tuple (IP address, port number).
- Each overlay node maintains a neighborhood table which contains a list of its neighbors in the overlay network. Each entry of a neighborhood table contains:
    - The logical address of the neighbor,
    - The physical address of the neighbor,
    - The time elapsed since the node last received a message from the neighbor.

Any protocol that builds an overlay network must provide mechanisms that enable nodes which are not members of the overlay to communicate with nodes in the overlay. These mechanisms, referred to as **rendezvous mechanisms**, are applied when new nodes join an overlay, and when when the overlay network has been partitioned and must be repaired. One can think of three rendezvous methods in an overlay network:

1. **Multicast:** non-members have a broadcast mechanism that is available to them (e.g. IP Multicast). They use this to announce themselves to members of the overlay network.
2. **Buddy List:** non-members maintain a list of members that are likely to be in the overlay network (a “buddy list”). They use this list to contact members.
3. **Server:** non-members contact a well-known server that establishes communication between members and non-members of an overlay network.

In HyperCast 2.0, the hypercube overlay network performs a rendezvous with multicast messages, and the Delaunay triangulation performs a rendezvous with the help of a server.

All overlay protocols that are used in HyperCast must be able to perform the following computation:

> *Given the logical address of some overlay node R, each overlay node with logical address A must be able to compute the logical address of A’s parent and child nodes in an embedded tree which has R as the root.*

With this ability nodes can perform unicast and multicast forwarding functions without the need for a routing protocol.

Overlay protocols in HyperCast are soft-state protocols. In soft-state protocols, all remote state information is periodically refreshed. If the remote state information is not refreshed, then it is invalidated. Timers are used to trigger the operations which recalculate and refresh the state information.

Hypercube (HC) Overlay Protocol[i]

In the Hypercube (HC) overlay protocol, members of the overlay are organized as the nodes of a logical *n-dimensional hypercube.*

An *n-dimensional hypercube* is a graph with 2^n nodes. Each node is labeled by a bit string $k_n \ldots k_1$, where $k_i \in \{0, 1\}$. Nodes in a hypercube are connected by an edge if and only if their bit strings differ in exactly one position. A hypercube of dimension $n = 3$ is shown in Figure 3.2.

The HC protocol works with incomplete hypercubes. In other words, the number of overlay members does not need to be a power of two.

3.1.1 Theory

Each node is identified by a label (e.g. "010"), which indicates the position of the node in the logical hypercube. In a hypercube, each node has only log(N) neighbors, where N is the total number of nodes. Also, the longest path in the hypercube is log(N).

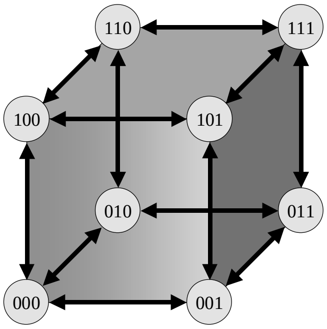


Figure 3.2. 3-dimensional hypercube with node labels.

It is relatively easy to embed trees in a hypercube topology. Recall that data transmission in the HyperCast overlays is done with trees that are embedded in the overlay network. A key idea that leads to the algorithm of building the embedded tree is to use a Gray code for ordering node labels of a hypercube. Another key idea is to add nodes to the hypercube in the order that is given by the Gray code. As an example, consider the labels of the 3-dimensional hypercube in Figure 3.2. If we want to add nodes to the hypercube, we need to have a rule for the order in which node labels are added. If we were to use the order of a binary encoding, then the nodes would be added in the following sequence: 000 → 001 → 010 → 011 → … → 111. However, if we use the order that is given by a Gray code, then we will add node labels in the following order: 000 → 001 → 011 → 010 → … → 100. Table 3.1 shows the ordering of labels according to a binary code and according to a Gray code. Note that consecutive node labels using a Gray code differ in exactly one bit position.

Table 3.1: **Comparison of a Binary code and a Gray code.**

Index *i* =	0	1	2	3	4	5	6	7
Binary code: *Bin(i)*	000	001	010	011	100	101	110	111

[i] Section 3.2 summarizes material from [BEAM99] and [LIEBE98b].

=								
Gray code: *G(i)=*	000	001	011	010	110	111	101	100

Using a Gray code, we can devise a simple algorithm which embeds a spanning tree in an incomplete hypercube. The algorithm, given in, implements a spanning tree in a distributed fashion. A node labeled G(i) calculates the label of its parent node in the tree with the root labeled G(r) by only using the labels G(i) and G(r) as input. The algorithm merely flips a single bit. The trees constructed by our algorithm have the following properties:

- **Property 1**. The path length between a node and a root is given by the Hamming distance of their labels.
- **Property 2.** If $N=2^n$, i.e. if the hypercube is complete, then the embedding results in a binomial tree.
- **Property 3.** In an incomplete and compact hypercube, the trees obtained by the algorithm are completely contained.

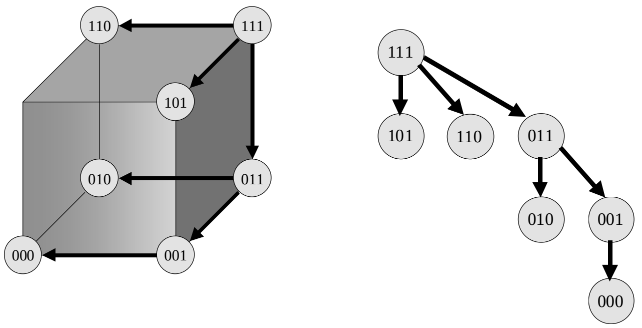
In Figure 3, We show the algorithm to calculate the parent of node *I* with respect to the embedded tree rooted at node *R*..

In Figure 4, we show a spanning tree that is generated by the algorithm for a root with label 111 in an incomplete hypercube with seven nodes.

Input: Label of the *I*-th node in the Gray encoding: $G(i) := I = I_n \ldots I_2 I_1$, and the label of the *r*-th *node* ($\neq i$) in the Gray encoding: $G(r) := R = R_n \ldots R_2 R_1$.
Output: Label of the parent node of node *I* in the embedded tree rooted at *R*.

```
Procedure Parent(I, R)
Begin
    If ($G^{-1}(I) < G^{-1}(R)$) {
        // Flip the least significant bit where I and R differ.
        Parent := $I_n I_{n-1} \ldots I_{k+1} (1 - I_k) I_{k-1} \ldots I_2 I_1$  with $k = \min_i (I_i \neq R_i)$
    }
    Else { // ($G^{-1}(I) > G^{-1}(R)$)
        // Flip the most significant bit where I and R differ.
        Parent := $I_n I_{n-1} \ldots I_{k+1} (1 - I_k) I_{k-1} \ldots I_2 I_1$ with $k = \max_i (I_i \neq R_i)$
    }
End
```

Figure 3.3. The Algorithm to build the embedded tree. $G^{-1}(.)$ is the inverse function of *G()* which assigns a number to a bit label, i.e. $G^{-1}(G(k)) = k$.



(a) Embedded in hypercube (b) Resulting tree

Figure 3.4. Embedded tree with node (111) as root.

3.1.2 Protocol Overview

Each overlay node in the hypercube has both a physical and a logical address. The physical address consists of the IP address of the host on which a node resides and the UDP port that is used by the node for HyperCast unicast messages. Each node has a unique physical address. The logical address of a node is a bit string label which uniquely indicates the position of the node in the hypercube. Logical addresses in the HyperCast protocol are represented as 32-bit integers, with one bit reserved to designate an invalid logical address. Therefore, the protocol allows for hypercubes of up to 231 (approximately two billion) nodes.

The task of the HC protocol is to keep the hypercube overlay network in a stable state. A stable state is one which is:

- **Consistent:** No two nodes share the same logical address.
- **Compact:** In a multicast group with N nodes, the nodes have bit string labels equal to G(0) through G(N – 1).
- **Connected:** Every node knows the physical address of each of its neighbors in the hypercube.

Nodes joining and leaving the hypercube and network faults can cause a hypercube to violate one or more of the above conditions. This results in an unstable state. The task of the HyperCast protocol is to continuously return the hypercube to a stable state in an efficient manner.

The HC protocol that is implemented in HyperCast uses IP multicast when new nodes join the node. This is done in order to prevent partitions. A node that wishes to participate in the hypercube first joins an IP Multicast group, referred to as the control channel. Every node can both send and receive messages on this channel. Obviously, the traffic on this channel should be kept minimal in order to comply with scalability requirements.

3.1.3 Ancestors, HRoot

Given any node, its successor in the Gray code ordering is defined to be its ancestor. In a stable hypercube, every node except the one with the largest logical address has one ancestor. A node without an ancestor is a Hypercube Root (HRoot). In the HyperCast protocol, every node keeps track of the logical address that is currently the highest in the hypercube according to the Gray code ordering. The node that has this logical address is assumed to be the HRoot. The address of the highest known logical address is used by a node to determine which of its neighbors should be present in its neighborhood table. If a node determines that a neighbor should be present in its neighborhood table and that neighbor is not present, then the node is said to have an incomplete neighborhood. Each node keeps the following information about the node with the highest logical address: the logical address, the physical address, the time elapsed since it last received a message from this node, and the last sequence number that was received from this node.

In an unstable hypercube, multiple nodes may consider themselves to be an HRoot. Also, different nodes in the hypercube may have different assumptions about which node is the HRoot. However, in a stable hypercube there is exactly one HRoot.

3.1.4 Hypercube Timers and Periodic Operations

Four time parameters are used in the Hypercube protocol. These parameters and their uses are defined below. Their default values are also listed.

$t_{heartbeat}$ (default = 2s): Nodes send messages to each of their neighbors in the neighborhood table every theartbeat seconds.

$t_{timeout}$ (default = 10s): When the time elapsed since a node last received a message from a neighbor exceeds ttimeout seconds, the neighbor’s entry is said to be stale and the neighborhood table is said to be incomplete. A missing neighbor is referred to as a tear in the hypercube. The information about the HRoot also becomes stale after ttimeout.

$t_{missing}$ (default = 20s): After the entry of one of its neighbors becomes stale, a node begins multicasting on the control channel to contact the missing neighbor. If the missing neighbor fails to respond for another tmissing seconds, then the node removes the entry from the neighborhood table and proceeds under the assumption that the neighbor has failed.

$t_{joining}$ (default = 6s): Nodes that are in the process of joining the hypercube send multicast messages to announce their presence to the entire group. A joining node that receives a multicast message from another joining node backs off from its attempt to join the hypercube for a period of time tjoining, before retrying to join. This prevents a large number of joining nodes from saturating the control channel with multicast messages.

3.1.5 Message Types

There are four message types that are used by the Hypercube protocol. All of these messages are sent as UDP datagrams. A node transmits a message by either unicasting to one or all of its neighbors or by multicasting on the control channel. We do not assume that the transmission of these messages is reliable.

Beacon Message: Beacon messages are multicast messages on the control channel. A beacon contains the logical/physical address pair of the sender, as well as the logical address of the currently known HRoot. A node transmits a beacon message only if it considers itself to be the HRoot, determines that it has an incomplete neighborhood, or is in the process of joining the hypercube.

Based on the construction of the hypercube, there is always at least one Hroot. Therefore, at least one node is able to send out beacons on the multicast channel. In a stable hypercube, there is only one Hroot. Thus, only one node sends out beacons to the multicast channel. Every node uses the beacon messages that are sent by HRoot or HRoots to form an estimate of the largest logical address in the hypercube. This information is sufficient for the node to determine whether it has a complete neighborhood.

Each beacon message contains a sequence number, SeqNo. This sequence number is used to resolve conflicts if beacons are received from multiple nodes. The HRoot's sequence number begins at zero. Whenever the HRoot sends a beacon message, the SeqNo is incremented by one. Whenever a new HRoot is chosen, the sequence number is also incremented (SeqNo of new HRoot = SeqNo of current HRoot + 1). Since each node keeps track of the current HRoot, the sequence number tracks the timeliness of the information on the HRoot. When information at a node is inconsistent, the information that is tagged with the lower sequence number is ignored.

The last group of nodes which send beacon messages are joining nodes which periodically send beacons to advertise their presence to the group.

Ping Message: Every node periodically sends a *ping* message to all of its neighbors that are listed in its neighborhood table. A ping informs the receiver that the node is still present in the hypercube. A *ping* is a short unicast message. It contains the logical and physical addresses of both the sender and the receiver of the message. It also contains the logical address and sequence number of the currently known *HRoot*. If a node has not received a *ping* from a neighbor for an extended period of time (t_{timeout}), then the node considers its neighborhood incomplete and begins sending *beacons* as described above. If it still has not received a *ping* from its neighbor after another period of time (t_{missing}), then it assumes that its neighbor has failed and removes the neighbor from its neighborhood list. Ping messages are also used as the only mechanism to assign a new logical address to the receiver of a *ping* message.

Leave Message: When a node wishes to leave the hypercube, it sends a *leave* message to its neighbors. When its neighbors receive this message, they will remove the node from their neighborhood tables. Since a leave message is not reliable, a node's neighbors may not always receive a leave messages when they should. In this case, the node's neighbors will notice its absence when it fails to respond to *ping* messages. Thus neighbors of a node that has left will eventually realize that it has left the neighborhood, even when they do not receive *leave* messages from it.

Kill Message: A *kill* message is used to eliminate a node from the hypercube. More specifically, a kill message is used to eliminate nodes with duplicate logical addresses. A node which receives a *kill* message immediately sends a *leave* message to all its neighbors and tries to rejoin the hypercube as a new node.

3.1.6 Protocol Mechanisms

The *HyperCast* protocol implements two mechanisms for maintaining a *stable* hypercube. Recall from Subsection 4.1 that a stable hypercube satisfies the criteria of being *consistent*, *compact*, and *connected*.

Duplicate Elimination *(Duel)*: The Duplicate Elimination (Duel) mechanism enforces consistency by ensuring that duplicate logical addresses are removed from the hypercube. If a node detects that another node has the same logical address, it compares its own physical address with the physical address of the conflicting node. If the node's physical address is numerically greater than the conflicting node's physical address, the node with the greater physical address issues a *kill* message to the other node. Otherwise, it sends *leave* messages to all of its neighbors and rejoins the hypercube.

Address Minimization *(Admin)*: The Address Minimization *(Admin)* mechanism is used to maintain compactness of the hypercube. On a conceptual level, the *Admin* mechanism has nodes attempt to assume lower logical addresses whenever opportunities to do so arise. To see how *Admin* reconstitutes compactness, recall that a hypercube which violates compactness must have a *tear* in the hypercube fabric (i.e. some node has an incomplete neighborhood table). The *Admin* mechanism enforces the rule that a node with a logical address that is higher than the logical address of the tear lowers its logical addresses to repair the tear.

The *Admin* mechanism at a node consists of an active and a passive part. The active part is executed when a node receives a *beacon* message from the *HRoot* and realizes that it is missing a neighbor which has a lower logical address than the *HRoot*. In such a situation, the node sends a *ping* with the missing lower logical address to the *HRoot*. The passive part is activated when the *HRoot* receives a *ping* message with a destination logical address that is lower than its current logical address. The *HRoot* sets its logical address to the value given in the *ping* in order to fix the hypercube.

The *Admin* mechanism also governs the process of nodes joining the hypercube. Initially, the logical address of a joining node is marked as invalid. The invalid address is larger than any valid address in the hypercube. Since a joining node sends *beacons* to announce its presence to the group, other nodes are able to check to see if they can find a lower (valid) logical address for the new node in the hypercube. If there is a node with an incomplete neighborhood, then this node will send a *ping* to the new node with the address of the vacant position. The new node assumes the (lower) address given in the *ping* message and occupies the vacant address. If there is no tear in the hypercube, then the new node is placed as a neighbor of the *HRoot*. More precisely, the *HRoot* sends a *ping* to the new node containing the logical address which corresponds to the successor of the *HRoot* in the Gray code ordering. Therefore, a node which joins a stable hypercube becomes the new *HRoot*.

The *Duel* and *Admin* mechanisms enforce, respectively, the consistency and compactness of a hypercube. The last criterion for a stable hypercube, connectedness, is maintained by the following process. Whenever a node *A* receives a message from a node *B* with a logical address that designates it as a neighbor in the hypercube, then the logical/physical address pair of node *B* is added into node *A*'s neighborhood table. If a node's neighbor does not send pings for an extended period of time, then the node will assumes that the neighbor has dropped out of the hypercube. As a result, it will remove that neighbor's entry in its neighborhood table. The Admin mechanism will then be used to repair the tear in the neighborhood table.

3.1.7 States and State Transitions

In the *HyperCa*st protocol, each node in the hypercube can be in one of eleven different states. Based on events that occur in the hypercube and *HyperCast* control messages that they receive, nodes transits between states. In Figure 3.5 we show the state transition diagram of the *HyperCast* protocol. The states are indicated as circles. State transitions are indicated as arcs, which are each labeled with a condition that triggers it. The possible states of the hypercube nodes are described in Table 2. With the state definitions, we can give a precise definition of a stable hypercube. A hypercube with *N* nodes is stable if all of its nodes have unique logical addresses ranging from *G*(0) to *G*(*N*-1) (where G(.) indicates the Gray code discussed in Section 3) and all of its nodes are in the *Stable* state, except for the node with the logical address *G*(*N*-1) which is in state *HRoot/Stable*.

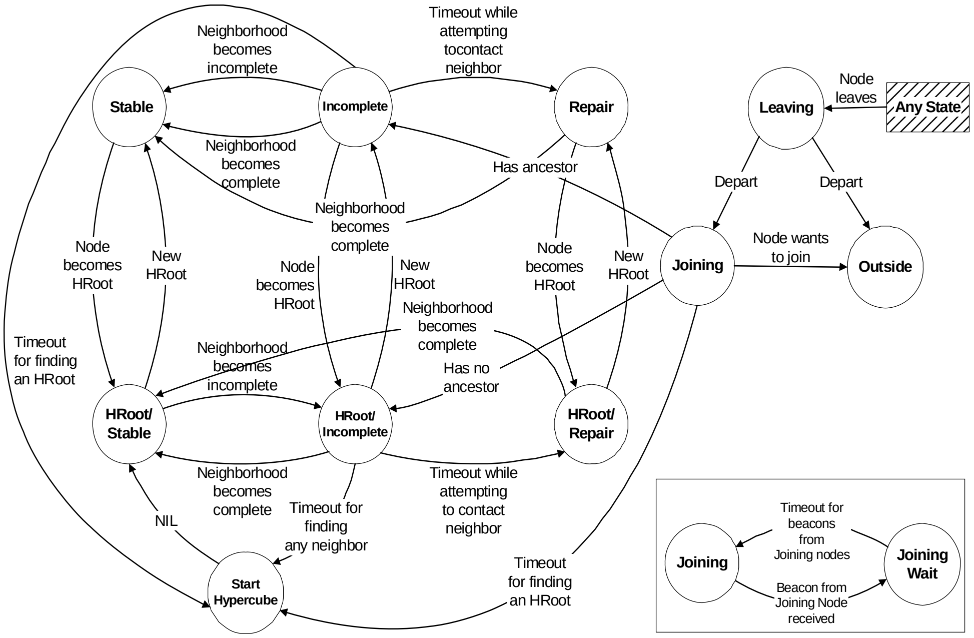

Neighborhood becomes incomplete
Timeout while attempting tocontact neighbor
Node leaves
Stable
Incomplete
Repair
Leaving
Any State
Neighborhood becomes complete
Has ancestor
Depart
Depart
Neighborhood becomes complete
Node becomes HRoot
New HRoot
Node becomes HRoot
New HRoot
Node becomes HRoot
New HRoot
Joining
Node wants to join
Outside
Neighborhood becomes complete
Timeout for finding an HRoot
Neighborhood becomes incomplete
Has no ancestor
HRoot/ Stable
HRoot/ Incomplete
HRoot/ Repair
Neighborhood becomes complete
Timeout for finding any neighbor
Timeout while attempting to contact neighbor
NIL
Start Hypercube
Timeout for finding an HRoot
Timeout for beacons from Joining nodes
Joining
Joining Wait
Beacon from Joining Node received

Figure 3.5. **Node state transition diagram.**

Table 3.2: **Node state definitions.**

Outside:	Not yet participating in the group.
Joining:	Wishes to join the hypercube, but does not yet have any information about the rest of the hypercube. Its logical address is marked as invalid.
JoiningWait:	A Joining node that has received a beacon from another Joining node within the last $t_{joining}$.
StartHypercube:	Has determined that it is the only node in the multicast group since it has not received any control messages for a period of time $t_{timeout}$. Starts its own stable hypercube of size one.
Stable:	Knows all of its neighbors' physical addresses.
Incomplete:	Either does not know one or more of its neighbors' physical addresses, or assumes that a neighbor has left the hypercube, because it has not received ping replies from that neighbor for $t_{timeout}$.
Repair:	Has been Incomplete for a period of time $t_{missing}$ and begins to take actions to attempt to repair its neighborhood.
HRoot/Stable:	Stable node which also believes that it has the highest logical address in the hypercube.
HRoot/ Incomplete:	Incomplete node which believes that it has the highest logical address in the entire hypercube.
HRoot/Repair:	Repair node which believes that it has the highest logical address in the hypercube.
Leaving:	Node that wishes to leave the hypercube.

3.1.8 Protocol Event Tables

The protocol actions that are taken by the nodes in response to events are presented in table form below. The "→" symbol means that the node will switch to the indicated state.

Table 3.3**: Event table for Outside state.**

Outside Node is not part of the hypercube

Event:	Action:
Application wants to join HyperCast group	→ Joining

Table 3.4**: Event table for Joining state.**

Joining

Wants to join the hypercube
Logical address is set as invalid

Event:	Action:
Periodically, every $t_{heartbeat}$	Send beacon message to control channel
No ping received for period $t_{timeout}$	→ StartHypercube
Beacon received from non-Joining node	Update known HRoot information
Beacon received from Joining node	→ JoiningWait
Ping received	Set own logical address to ping's destination logical address
After ping received, own logical address equals known HRoot's logical address	→ HRoot/Incomplete
After ping received, own logical address does not equal known HRoot's logical address	→ Incomplete

Table 3.5: **Event table for JoiningWait state.**

JoiningWait

Wants to join the hypercube
Has received a beacon from a Joining node
Logical address is set as invalid

Event:	Action:
No ping received for period $t_{timeout}$	→ StartHypercube
Beacon received from non-Joining node	Update known HRoot information
No beacon received from Joining node for period $t_{joining}$	→ Joining
Ping received	Set own logical address to ping's destination logical address
After ping received, own logical address equals known HRoot's logical address	→ HRoot/Incomplete
After ping received, own logical address does not equal known HRoot's logical address	→ Incomplete

Table 3.6: **Event table for StartHypercube state.**

StartHypercube Start new hypercube

Event:	Action:
	Set own logical address to G(0) → HRoot

Table 3.7: **Common event table for several states.**

Stable

Incomplete

Repair

HRoot/Stable

HRoot/Incomplete

HRoot/Repair

Event:	Action:
Periodically, every $t_{heartbeat}$	Send ping message to all valid neighbors
Application triggers leave	Send leave message to all valid neighbors → Leaving
Receive message with source logical address equal to own logical address and source physical address less than own	Send kill to message source
Receive message with source logical address equal to own logical address and source physical address is greater than own	Send leave to all valid neighbors → Leaving
Kill received	Verify that source's physical address is greater than own physical address Send leave to all valid neighbors → Leaving
Ping received	Update neighborhood entry for sender's logical address Update known HRoot information
Beacon received	Update known HRoot information
Leave received	Remove neighborhood entry for sender's logical address

Table 3.8: **Event table for Stable state.**

Stable

Event:	Action:
Neighborhood becomes incomplete due to lack of pings from a neighbor for period $t_{timeout}$	→ Incomplete
Own logical address is greater than known HRoot logical address	→ HRoot/Stable

Table 3.9: **Event table for Incomplete state.**

Incomplete

Event:	Action:
Periodically, every $t_{heartbeat}$	Send beacon message to control channel
Neighborhood becomes complete	→ Stable
Own logical address is greater than known HRoot logical address	→ HRoot/Incomplete
Neighborhood partially empty for timeout interval $t_{missing}$	→ Repair
Neighborhood completely empty	→ StartHypercube

Table 3.10: **Event table for Repair state.**

Repair

Event:	Action:
Periodically, every $t_{heartbeat}$	Send beacon message to control channel
Beacon received from HRoot or Joining node	Send ping message to beacon source containing new logical address to fill tear in neighborhood
Neighborhood becomes complete	→ Stable
Own logical address is greater than known HRoot logical address	→ HRoot/Repair
Neighborhood completely empty	→ StartHypercube

Table 3.11: **Event table for HRoot/Stable state.**

HRoot/Stable

Event:	Action:
Periodically, every $t_{heartbeat}$	Send beacon message to control channel Increment sequence number
Beacon received from Joining node	Register Joining node as next higher neighbor Increment sequence number Update known HRoot information to be new HRoot
Neighborhood becomes unstable due to lack of pings from a neighbor for period $t_{timeout}$	→ HRoot/Incomplete
Own logical address is less than known HRoot logical address	→ Stable

Table 3.12: **Event table for HRoot/Incomplete state.**

HRoot/Incomplete

Event:	Action:
Periodically, every $t_{heartbeat}$	Send beacon message to control channel Increment sequence number
Beacon received from Joining node	Register Joining node as next higher neighbor Increment sequence number Update known HRoot information to be new HRoot
Neighborhood becomes complete	→ HRoot/Stable
Own logical address is less than known Hroot logical address	→ Incomplete
Neighborhood partially empty for timeout interval $t_{missing}$	→ HRoot/Repair
Neighborhood completely empty	→ StartHypercube

Table 3.13: **Event table for HRoot/Repair state.**

HRoot/Repair

Event:	Action:
Periodically, every $t_{heartbeat}$	Send beacon message to control channel Increment sequence number
Beacon received from Joining node	Send ping message to beacon source containing new logical address to fill tear in neighborhood
Neighborhood becomes complete	→ HRoot/Stable
Own logical address is less than known Hroot logical address	→ Repair
Neighborhood completely empty	→ StartHypercube

Table 3.14: **Event table for Leaving state.**

Leaving	Waits for period $t_{timeout}$ to ensure that neighbors receive leave messages in response to their pings Proceeds to Outside if leave was initiated by application, otherwise proceeds to Joining

Event:	Action:
Ping received	Send leave to message source
Leave was triggered by application and $t_{timeout}$ time has elapsed	→ Outside
Leave was not triggered by application and $t_{timeout}$ time has elapsed	→ Joining

3.1.9 Message Packet Format

Basic messages are sent using the following packet format, which is common to all messages:

1 byte	Message Type
4 bytes	Source IP Address
4 bytes	Source Port
4 bytes	Source Logical Address
4 bytes	Dest IP Address
4 bytes	Dest Port
4 bytes	Dest Logical Address
4 bytes	HRoot Logical Address
4 bytes	HRoot Sequence Num
4 bytes	Data Length
- bytes	Data

Figure 3.6. **Packet format.**

The Message Type field is defined as follows:

Table 3.15: **Message types.**

Message Type:	Ping	Beacon	Leave	Kill
Field Value:	0	1	2	3

The IP Address fields are filled in the network address' most significant byte to least significant byte order. The Port, Logical Address, Sequence Number, and Data Length fields are also filled in order from the most significant to the least significant byte.

Data is a variable-length field, with its length specified by the Data Length field of the packet. The Data field is present for future expansion, and it is not currently used in the protocol.

3.1.10 Example

We next illustrate the operations of the protocol in a simple example. In this example, we use a small number of nodes and assume that there are no packet losses.

Figure 3.7 shows a hypercube with five nodes, which are represented as circles. We use arrows to represent unicast messages. Circles around a node indicate a multicast message. In Figure 3.7-a, we show a stable hypercube. Here, the *HRoot*, node *110*, multicasts *beacons* periodically. The *beacon* is received by all nodes and keeps them informed of the logical address of the *HRoot*. Therefore, the nodes know which of their neighbors should be present in their neighborhood tables. Every node periodically sends *ping* messages to its neighbors in the neighborhood table (Figure 3.7-b).

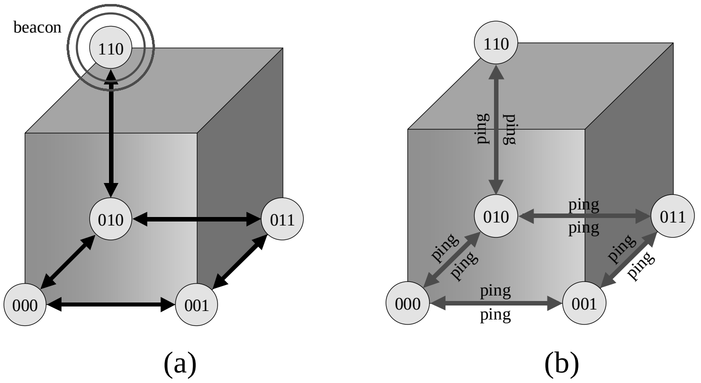


Figure 3.7. **Stable hypercube.**

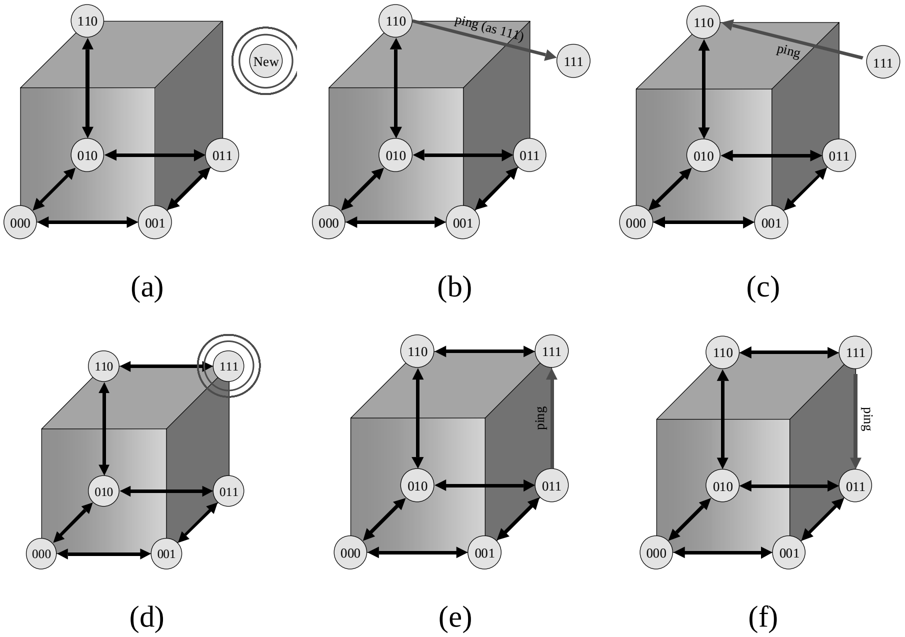
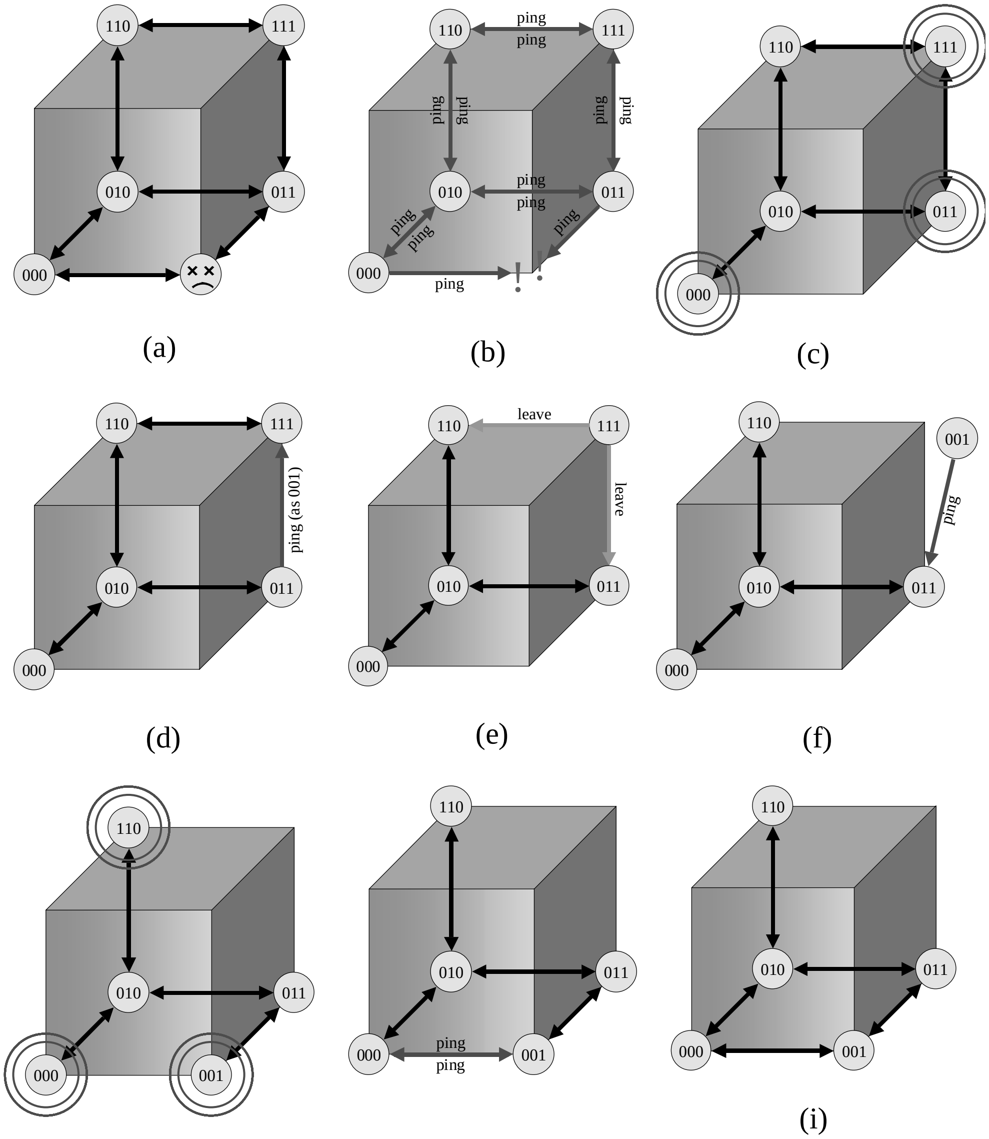


Figure 3.8. **Joining node.**

In Figure 3.8-a**,** we show a node in the *Joining* state. The node is labeled "New" and wants to join the hypercube. The node periodically sends *beacon* messages in order to make its presence known to the group. The *HRoot* places the *Joining* node as its neighbor in the next successive position in the hypercube, according to the Gray code ordering. Then, it *pings* the new node with the new logical address (*111*) (Figure 3.8-b). The new node takes on the new logical address and replies with a *ping* back to the original *HRoot* (Figure 3.8-c). The new node determines from the *ping* packet that it is the *HRoot*, since its own logical address is the highest known logical address. It begins sending *beacons* as an *HRoot* (Figure 3.8-d). If node *011* receives the *beacon* from the new *HRoot*, then it realizes that *111* should be its neighbor. Thus, node *011* sends a *ping* message to *111* (Figure 3.8-e). Once node *111* receives the *ping* message, it responds with a *ping* itself (Figure 3.8-f). At this time, all of the nodes in the hypercube have complete neighborhood tables and know all their neighbors, so the hypercube is stable.

3.1.11 Repairing a Tear

The process of repairing defects in the hypercube control topology is shown here.



(a) (b) (c)

(d) (e) (f)

(i)

(g) (h)

Figure 3.9. **Repairing a tear.**

It is possible for a node to fail unexpectedly (Figure 3.9-a). Nodes that have failed are detected when their neighbors do not receive *ping* messages from them for a period of time t_{timeout} (Figure 3.9-b). Each of the failed node's neighbors then periodically sends *beacons* to indicate that they have detected a missing neighbor (Figure 3.9-c). Note that if the failed node returns at this time, the *beacons* from its neighbors will be used to reestablish the logical connections in its neighborhood table. After sending *beacons* for a period of time t_{missing} without receiving a reply, each neighbor assumes that the failed node will not return and a replacement is needed. The *Admin* mechanism then begins as one or more neighbors send a *ping* to the HRoot. This is done in order to lower the HRoot's logical address and fill the tear (Figure 3.9-d).

Upon receiving the *ping*, the HRoot sends *leave* messages to its neighbors to notify them that the HRoot will be leaving their neighborhoods (Figure 3.9-e). The HRoot then assumes the new logical address that was given to it by the failed node's neighbor. It replies to the failed node's neighbor with a *ping* of its own (Figure 3.9-f). This completes the logical connection between the two nodes, since both nodes have entries for each other in their respective neighborhood tables and know each other's physical addresses. The relocated HRoot then *beacons*, since it does not yet know all of its neighbors (Figure 3.9-g). The neighboring nodes receive each other's *beacons* and respond by sending *pings* (Figure 3.9-h). This completes the repair procedure and the hypercube returns to a stable state (Figure 3.9-i).

3.1.12 Evaluation and Discussion

We used the **Spin** protocol verification tool [HOLZ97] to aid in the development of the *HyperCast* Protocol. *Spin* checks the logical consistency of a protocol specification by searching for deadlocks, non-progress cycles, and any kind of violation of user-specified assertions. To verify the *HyperCast* design in *Spin*, the entire *HyperCast* protocol specification, as well as a system for simulating multiple hypercube nodes was encoded using the Process Meta Language (*PROMELA*). In addition to checking for deadlocks and non-progress cycles, *Spin* was used to ensure that every execution path resulted in a stable hypercube.

Due to the unavoidable state space explosion when using a tool such as *Spin*, we were only able to analyze hypercubes with at most six nodes. While verification cannot be used to prove results for large hypercube sizes, we assert that for the purposes of verification there is little qualitative difference between a hypercube of six nodes and a hypercube of several thousand nodes. It is unlikely that non-progress cycles and deadlocks will exist in large hypercubes that do not have analogous fault modes in a six node hypercube. However, we wish to emphasize that our verification with *Spin* is not equivalent to a complete formal verification of the protocol.

The HyperCast protocol has been run on a Linux Cluster with up to 10,000 nodes [BEAM99][LORIN01]. Overall, the hypercube takes a long time to stabilize if the number of nodes is large. Since the HC protocol always enforces the rule that the lowest positions of the hypercube (according to the Gray code ordering) are occupied, the addition of nodes is serialized. This slows down the process of adding many nodes.

The mapping of the overlay network to an underlying network has been evaluated in [LIEBE01a].

Delaunay Triangulations[ii]

A Delaunay triangulation is special type of triangulation: for each circumscribing circle of a triangle formed by three nodes, no other node of the graph is in the interior of the circle. Each node in a Delaunay triangulation has (x,y) coordinates which depict a point in the plane.

An advantage of the Delaunay triangulation is that it can be constructed in a distributed fashion (see [LIEBE01a]). Therefore, Delaunay triangulations can be built very quickly. In a triangulation, each node has an average of six neighbors. however, in the worst-case , a node can has N-1 neighbors where N is the total number of nodes.

If the (x,y) coordinates of a node in the Delaunay triangulation reflect its geographical location, then nodes in the overlay network are likely to be neighbors if their geographical locations are close. However, a Delaunay triangulation is not aware of the layer-3 network infrastructure.

3.1.13 Delaunay Triangulation as an Overlay Network Topology

A Delaunay triangulation for a set of vertices A is a triangulation graph with the defining property that for each circumscribing circle of a triangle formed by three vertices in A, no vertex of A is in the interior of the circle. In Figure 3.10, we show a Delaunay triangulation and the circumscribing circles of some of its triangles. Delaunay triangulations have been studied extensively in computational geometry and have been applied in many areas of science and engineering, including communication networks.

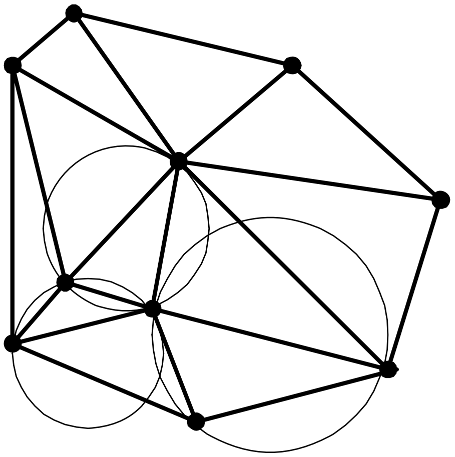


Figure 3.10. **A Delaunay Triangulation.**

3.1.14 Delaunay Triangulation Overlay Network

In order to establish a Delaunay triangulation overlay, each application (node), is associated with a vertex in the plane with given (x,y) coordinates. The coordinates are assigned via some external mechanisms (e.g. GPS or user input) and can be selected to reflect the geographical locations of nodes. Two nodes have a logical link in the overlay,

[ii] Section 3.2 summarizes material from [LIEBE01b].

i.e. are neighbors, if their corresponding vertices are connected by an edge in the Delaunay triangulation. The Delaunay triangulation has several properties that make it attractive as an overlay topology for application-layer multicast. First, it normally has different, non-overlapping routes between any pair of vertices. The existence of such different paths can be exploited by an application-layer overlay when nodes fail or are not responsive. Second, the number of edges at a vertex in a Delaunay triangulation is generally small. Specifically, since each triangulation of n vertices has at most *3n-3* edges, the average number of edges at each vertex is less than six. Despite the worst-case in which the number of edges at a vertex is *n-1*, the maximum number of edges is usually small. Third, once the topology is established, packet forwarding information is encoded in the coordinates of a node. Thus, there is no need for a routing protocol. Finally, the Delaunay triangulation can be established and maintained in a distributed fashion. We elaborate on the last two properties in the next subsections.

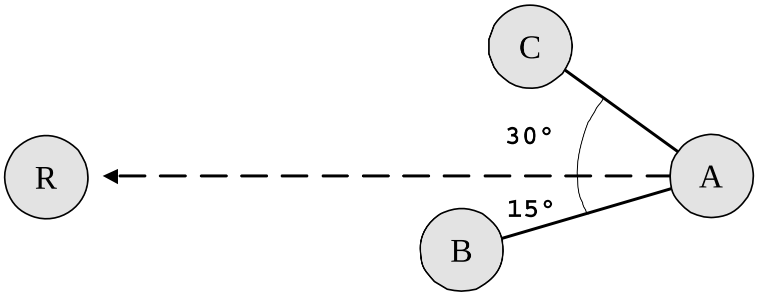


Figure 3.11. Compass Routing. Node A has two neighbors, B and C. A computes B as the parent in the tree with root R, since the angle ∠RAB = 15º is smaller than the angle ∠RAC=30º.

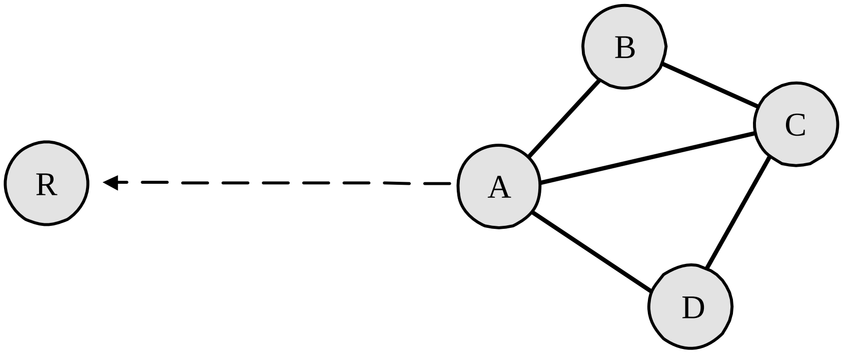


Figure 3.12. Compass Routing. Node A determines that it is the parent for node C, since the angle ∠RCA is smaller than angles ∠RCD and ∠RCB. Likewise, B and D determine that they are not the parents of node C, since ∠RCA < ∠RCB and ∠RCA < ∠RCD.

3.1.15 Compass Routing

Multicast and unicast forwarding in the Delaunay triangulation is done along the edges of a spanning tree that is embedded in the Delaunay triangulation overlay. The tree that has the sender as its root. In the Delaunay triangulation, each node can locally determine its child nodes with respect to a given tree by using its own coordinates, the coordinates of its neighbors, and the coordinates of the sender.

Local forwarding decisions at nodes are made by using compass routing [KRANA99]. The basic building block of compass routing is that a node A, for a root node R, computes a node B as its parent in the tree, if B is the neighbor with the smallest angle to R. This is illustrated in Figure 3.11. Compass routing is also used for determining a multicast routing tree, where nodes calculate their child nodes in the multicast routing tree in a distributed fashion. Specifically, a node A determines that one of its neighbors C is a child node with respect to a tree with root R. In order to determine this, A uses the following considerations. Since the overlay topology is a triangulation, the edge AC is a border of two triangles, say ∇ABC and ∇ACD (see Figure 3.12). A determines that C is a child node with respect to R, if selecting A leads to a smaller angle from C to R, than selecting B and D. If each node performs the above steps for determining child nodes, then the nodes compute a spanning tree with root node R.

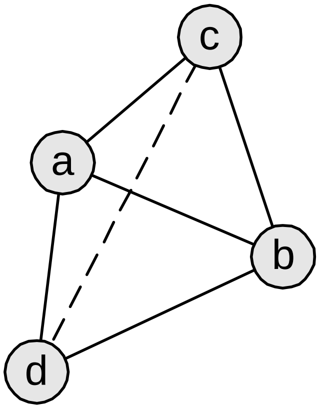


Figure 3.13. Locally equiangular property. The property holds for triangles ∇abc and ∇abd if the minimum internal angle is at least as large as the minimum internal angle of triangles ∇acd and ∇cdb.

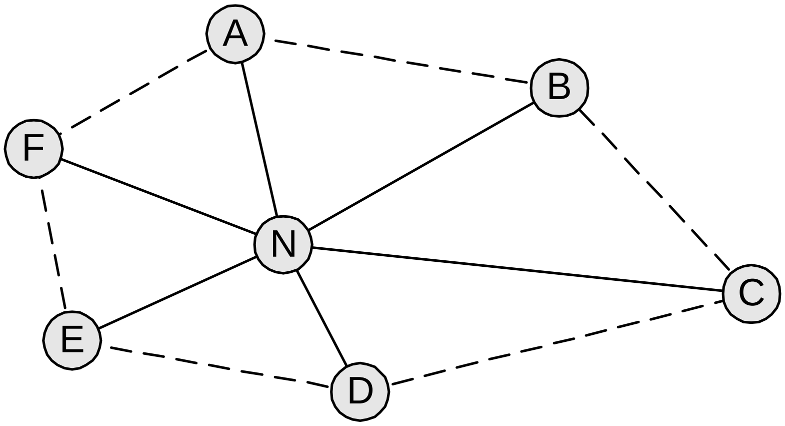


Figure 3.14. Locally equiangular property for a node N and its neighbors in the graph. Node N can enforce the equiangular property for all quadrilaterals that are formed by N and its neighbors A, B, C, D, E, and F. Here, N detects that the locally equiangular property is violated for triangles ∇NBC and ∇NCD. Thus, the edge $\overline{NC}$ should be replaced by edge $\overline{DB}$.

3.1.16 Building Delaunay Triangulations with Local Properties

Delaunay triangulations can be defined in terms of a locally enforceable property, which is illustrated in Figure 3.13. A triangulation is said to be *locally equiangular* if, for every quadrilateral that is formed by triangles ∇acb and ∇abd (which share a common edge, in this case it is $\overline{ab}$) the minimum internal angle of triangles ∇acb and ∇abd is at least as large as the minimum internal angle of the triangles ∇acd and ∇cbd. In [SIBS77], it was shown that a locally equiangular triangulation is a Delaunay triangulation.

In a graph that is a triangulation, each node N can enforce the locally equiangular property for all quadrilaterals that are formed by N and its neighbors. In Figure 3.14, node N can detect that the locally equiangular property is violated for triangles ∇NBC and ∇NCD. It can also detect that the edge $\overline{NC}$ should be removed and replaced by an edge $\overline{DB}$. Thus, N can remove node C from its list of neighbors.

The protocol described in the next section builds and maintains a Delaunay triangulation overlay by enforcing the locally equiangular property for each node and its neighbors.

3.1.17 Overview of the DT Protocol

We next describe the network protocol which establishes and maintains a set of applications in a logical Delaunay triangulation. Essentially, the network protocol implements a distributed incremental algorithm for building a Delaunay triangulation.

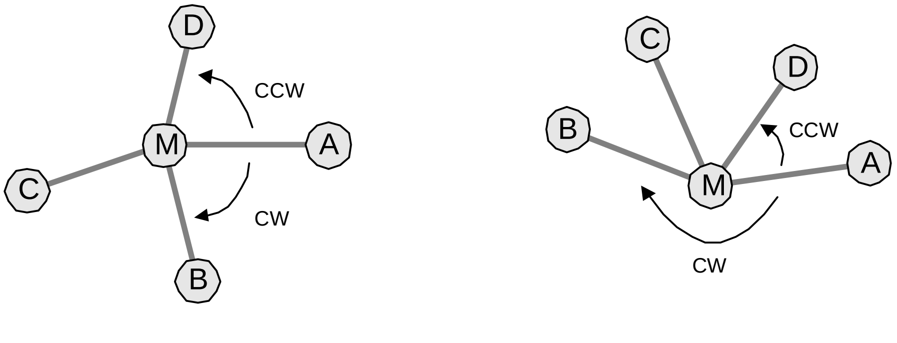
In the following, we will refer to the protocol entities that execute the DT protocol as *nodes*. Each node has a logical address and a physical address. The logical address of a node is represented by (x,y) coordinates in a plane, which identify the position of a vertex in a Delaunay triangulation. We set the x and y coordinates to lengths of 32 bits each. The logical address of a node is a configuration parameter. It can be either assigned to a node or derived from the geographical location of the IP address of a node. The physical address of a node is a globally unique identifier on the Internet, consisting of an IP address and a UDP port number.

We will denote the coordinates of a node A as coord(A)=(x_A, y_A). We define an ordering of nodes where coord(A) < coord(B), if $y_A < y_A$, or $y_A = y_B$ and $x_A < x_B$.

3.1.18 Neighbors and Neighbor Test

We say two nodes are neighbors if the edge connecting the two nodes appears in the Delaunay Triangulation graph. Each node maintains a *neighborhood table* which contains its neighbors in the Delaunay Triangulation overlay.

The protocol operations at a node mainly consists of adding and removing neighbors in its neighborhood table. To add or remove another node to or from its neighborhood table, a node needs to know if that node is eligible to be its neighbor in the current topology. We next describe the *neighbor test* algorithm we developed for this purpose. This *neighbor test* is mainly based on the *locally equiangular* property described in the previous subsections.



(a) CW angle < 180 (b) CW angle >=180

Figure 3.15. **CW and CCW neighbors**

Before describing this algorithm, we first give the definitions of the clockwise (CW) and counter-clockwise (CCW) neighbors of a given node, say node A, with respect to another node, say node B, since these two concepts are very important in the *neighbor test* algorithm. A neighbor of node A is said to be CW or CCW neighbor with respect to node B, if (1) it forms the smallest CW or CCW angle to node B taking node A as the pivot and (2) the smallest CW or CCW angle is less than 180 degrees. The notions of CW and CCW neighbors are illustrated in Figure 15, in which we consider node M with respect to node A.

In Figure 3.15(a), node B is the clockwise neighbor of M with respect to A. While in Figure 3.15(b), since the CW angle is larger than 180 degrees, node B will not be regarded as the clockwise neighbor. In this case, we say node M has no CW neighbor with respect to node A. In both Figure 3.15(a) and (b), node D is the counter-clockwise neighbor of M with respect to node A.

We now describe the *neighbor test* algorithm. In the neighbor test, a testing node determines if another (the tested) node should or should not be its neighbor. The testing node performs the neighbor test by looking at the coordinates of its current neighbors and the tested node. The test covers all possible locations of the tested node, relative to the testing node and the neighbors of the testing node.

In the following description, M denotes the testing node and A denotes the tested node. Essentially, the neighbor test verifies the *locally equiangular* property for convex quadrilaterals from Subsection 2.3. That is, if M has CW and CCW neighbors with respect to A, and the quadrilateral formed by M, A, and these two neighbors is convex, A passes the neighbor test at M, if the edge MA maximizes the minimum internal angle. Otherwise, A does not pass the neighbor test at M.

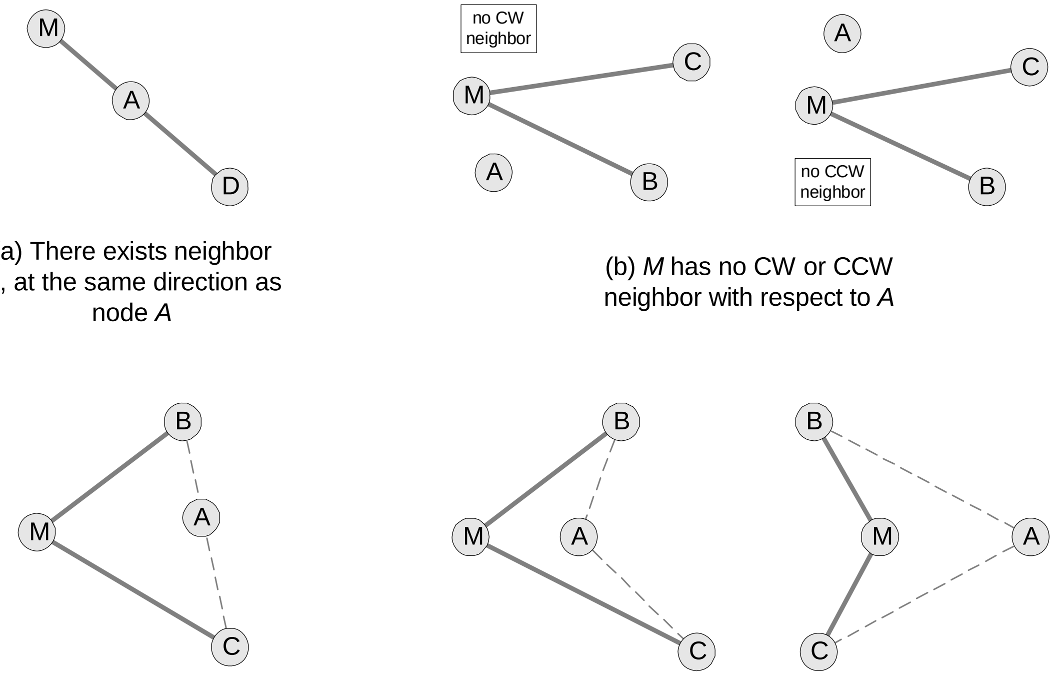
However, there are several cases to consider where the above test can not be made. In these cases, A passes the neighbor test at M, if adding A results in a triangulation. The following is a complete set of all feasible cases:

1. If A has a neighbor D, such that M, A, and D lie on the same line, A passes the neighbor test, if A is closer to M than D. This is illustrated in Figure 3.16(a).

2. If M does not have a CW or a CCW neighbor with respect to A, A passes the neighbor test. This is illustrated in Figure 3.16(b). Note

that this includes the case where M has neither a CW nor a CCW neighbor with respect to A.

3. If the quadrilateral formed by M, the CW and CCW neighbors, and A degenerates to a triangle (see Figure 3.16(c)) or it is concave (see Figure 3.16(d)), A passes the neighbor test at M.



(a) There exists neighbor *D*, at the same direction as node *A*

(b) *M* has no CW or CCW neighbor with respect to *A*

(c) The quadrilateral degenerates to a triangle

(d) A concave quadrilateral is formed

Figure 3.16**. Neighbor test**

As described above, M rejects A only in two cases: (1) There is already a neighbor of the testing node on the same direction as the tested node, and that neighbor is closer to the testing node. (2) The *locally equiangular* property is violated in the convex quadrilateral found out. For all other cases, M will accept A.

We can argue the correctness of the neighbor test as follows. The neighbor test is a consequence of the locally equiangular property from [SIBSON77], which states that a triangulation where all convex quadrilaterals are locally equiangular is a delaunay triangulation. The neighbor test enforces the property from [SIBSON 77] by enforcing two points: (1) Whenever a convex quadrilateral is formed by M, A, and the CW and CCW neighbors, then the locally equiangular property is enforced; (2) When no convex quadrilateral can be formed by M, A, and the CW and CCW neighbors, i.e., the locally equiangular property is not applicable, then node A passes the neighbor test at M if adding A as a neighbor forms a triangulation.

Each node periodically sends *neighbor messages* to nodes in its neighborhood table. A neighbor messages contains the physical and logical addresses of the sending node, as well as the logical and physical addresses of its CW and CCW neighbors with respect to the receiver.

When node M receives a neighbor message which contains the addresses of A, $CW_M(A)$, and $CW_M(A)$, it updates its neighborhood table as follows:

Neighborhood table at M:

Neighbor	CW Neighbor	CCW Neighbor
A	E = CW_M (A)	F = $CW_M(A)$
…	…	…

Each node that sends a neighbor message to node M and passes M's neighbor test is added as a neighbor in M's neighborhood table. If a node in the neighborhood table fails the neighbor test, it is removed from neighborhood tables. If node A knows the logical and physical addresses of some node X that is not a neighbor of A and if X passes the neighbor test for A, then X becomes a candidate neighbor at A. Node A can learn about the existence of other nodes through messages that it receives, which are not neighbor messages, and from the CW or CCW neighbor columns in its neighborhood table (e.g. E and F in the above table). If a node has candidate neighbors, it will send them neighbor messages. If a candidate neighbor responds with a neighbor message, it is elevated to the status of a neighbor. If two nodes are neighbors of each other, then this represents a link in the triangulation.

A neighbor can be removed from the neighborhood table for any one of the following reasons: (1) the neighbor has sent a message indicating that it has left the overlay, (2) no message has been received from this neighbor for an extended period of time, or (3) the neighbor has failed a neighbor test.

3.1.19 DT Servers and Leaders

Any protocol that builds an overlay network must provide mechanisms that enable nodes which are not members of the overlay to communicate with nodes in the overlay. These mechanisms are applied when new nodes join an overlay and when the overlay network has been partitioned and must be repaired.

In the DT protocol, a server facilitates the addition of members to the overlay and the partitioning of overlay networks. A reservation against using a well-known server is that the server may become a performance bottleneck. However, in our experiments a single server was sufficient to manage the workload from 10,000 new members joining the overlay in a short period of time. Another potential problem with well-known servers is that they provide likely points of failure. We emphasize that variations of the DT protocol which use broadcast announcements or buddy lists can be derived without changing the main characteristics of the DT protocol. Also, one can adapt the protocol to support multiple servers.

The server component of the DT protocol is called the DT server. New nodes join the overlay network by sending requests to the DT server. The server responds with the logical and physical addresses of some node that is already in the overlay network. The new node then sends a message to the node identified by the DT server. Thus, it has established communication with some node in the overlay network.

The DT server is also engaged in repairing partitions of the overlay network. In the DT protocol, a node believes that it is a *Leader* if it does not have a neighbor with a greater logical address (using the ordering given at the beginning of this section). Each Leader periodically sends messages to the DT server. If an overlay has a partition, then more than one node will believe that they are Leaders. If the server receives messages from multiple Leaders, then it replies with the identity of the Leader with the greatest coordinates. If a node A believes it is a Leader and learns about a node B with coord(A) < coord(B), then B will pass A's neighbor test. Consequently, A will add node B as a candidate neighbor and will no longer believe that it is a Leader.

Each DT server maintains a list (cache) of the logical and physical addresses of other nodes in the overlay. When the DT server sends the address of a node in the overlay to a node that is trying to join, the address is taken from the cache. We set the default size of the cache to 100 nodes. The DT server periodically queries nodes in the cache to verify that they are still members of the overlay. If a node does not respond to a query, then it will eventually be removed from the cache. Also, a node is removed from the cache when the DT server has selected this node six times as the contact node for a node that is trying to join. If a node that is trying to join contacts the DT server and the cache is not full, then it will be added to the cache.

3.1.20 Timers

The DT protocol is a soft-state protocol. This means that all remote state information is periodically refreshed. If it is not refreshed, then it is invalidated. The operations that recalculate and refresh state are triggered by timers. A node of the DT protocol uses the following three timers.

- **Heartbeat Timer.** The heartbeat timer determines when a node sends messages to its neighbors. The timer runs in two modes, SlowHeartbeat and FastHeartbeat. A node is in FastHeartbeat mode when it joins the overlay and when it has candidate neighbors. In all other cases, the node is in SlowHeartbeat mode. The operation of the heartbeat timer in two modes attempts to trade off the need for fast convergence of the overlay network when the topologies change and low bandwidth consumption in a steady state. In our experiments, we set the heartbeat timer to $t_{SlowHeartbeat}$= 2 seconds in SlowHeartbeat mode and to $t_{FastHeartbeat}$= 0.25 seconds in FastHeartbeat mode.
- **Neighbor Timers.** Each node deletes neighborhood table entries that are not refreshed within $t_{Neighbor}$ seconds. Also, a DT server invalidates its cache entries and the information on the Leader(s) after $t_{Neighbor}$ seconds. We set the value of the Timeout timer to $t_{Neighbor}$= 10 seconds.
- **Backoff Timer.** When a node does not receive a reply from the DT server, it retransmits its request using an exponential back-off algorithm with a Backoff timer. Initially, the Backoff timer is set to $t_{FastHeartbeat}$ and doubled after each repeated transmission. It does this until it reaches $t_{Neighbor}$= 10 seconds. If there are alternate DT servers, then the node switches to an alternate DT server when $t_{Neighbor}$= 10 seconds.

The DT server uses the following two timers:

Cache Timer. If the DT server has not received a CachePong message from a node in its node cache, in response to CachePing message for t_{Cache} seconds, the node will be deleted from the cache. There is, however, one exception. The node cache entry for the node with the largest coordinates, the Leader, is not deleted, even if the the cache timer expires. There is one cache timer for each node in the node cache. The default timeout value of the timer is t_{Cache}=10 seconds.

Leader Timer. If the DT server has not received a message from the Leader for t_{Leader} seconds, another node from the node cache will be selected as Leader. The default timeout value of the leader timer is t_{Leader} seconds.

3.1.21 Message Types

The DT protocol has eight types of messages, which are sent as UDP datagrams. All DT protocol messages are sent as unicast messages. We describe the contents of each message and the operations associated with the transmission and reception of each message.

- **HelloNeighbor and HelloNotNeighbor Messages.** These messages are used to create and refresh neighborhood tables at nodes. Each *HelloNeighbor* and *HelloNotNeighbor* message contains the logical and physical addresses of the sender and the clockwise and counter-clockwise neighbors of the sender with respect to the receiver. Each time the Heartbeat timer goes off, a node sends *HelloNeighbor* messages to each of its neighbors and to one of its candidate neighbors, if there is a candidate neighbor. If there are multiple candidate neighbors, then the message is sent to the candidate neighbor with the closest coordinates.

  HelloNotNeighbor messages are sent as immediate replies to *HelloNeighbor* messages from nodes that fail the neighbor test. The *HelloNotNeighbor* message serves three purposes. First, the information in the message is used by the receiver to update its neighborhood table. Second, the clockwise and counter-clockwise neighbors in the *HelloNotNeighbor* message provide the receiver with additional information about neighbors in its vicinity. Lastly, *HelloNotNeighbor* messages are used to resolve situations where two nodes have the same logical address.

- **Goodbye Message.** When a node leaves the overlay, it sends *Goodbye* messages to the DT server and to all of its neighbors. If a node receives a *Goodbye* message, then it removes the sender of the *Goodbye* message from its neighborhood table. The DT server removes the sender of a *Goodbye* message from its cache. A node that has sent *Goodbye* messages can continue to send *Goodbye* messages in response to messages that it receives, until the process that runs the node is terminated by the application.

- **ServerRequest and ServerReply Messages.** *ServerRequest* and *ServerReply* messages are, respectively, queries to and replies from the DT server. *ServerRequest* messages are sent by nodes that are trying to join and by Leaders. A Leader sends a *ServerRequest* message every $t_{FastHeartbeat}$ seconds. *ServerRequest* messages are retransmitted if no *ServerReply* is received, using the exponential backoff outlined above.

  Each *ServerRequest* message contains the logical and physical addresses of the sender. The *ServerReply* message contains the logical and physical addresses of some node in the overlay. More specifically, a *ServerReply* sent to node X contains the logical and physical addresses of some node Y, with X ≤ Y.

Newly joining nodes use addresses in the ServerResponse message to find a node that is already in the overlay. Leaders use the addresses in the ServerResponse message to determine if the overlay has a partition.

- **NewNode Message.** The *NewNode* message contains the logical and physical addresses of a new node. When a new node N obtains the address of some node in the overlay D from the DT server, then N will send a *NewNode* message to D. If N passes the neighbor test at D, then N becomes a candidate neighbor at D, and D responds to N with a *HelloNeighbor* message. Otherwise, D passes the *NewNode* message to one its neighbors whose coordinates are closer to those of N. Thus the *NewNode* message is routed through the overlay toward the coordinates of the new node, until the *NewNode* message reaches a node where the new node passes a neighbor test.
- **CachePing and CachePong Messages.** CachePing and CachePong messages are used to refresh the contents of the cache at the DT server. Every $t_{SlowHeartbeat}$ seconds, the DT server sends a CachePing message to each node in the cache. A node that receives a CachePing message immediately replies with a CachePong message.

3.1.22 Shifting Coordinates

Since the logical address of a node is a configuration parameter, it is possible for two nodes to have the same coordinates. It is also possible for the coordinates of four nodes to lie in a circle (created by three nodes). In the former case, the Delaunay triangulation is not defined. In the latter case, the Delaunay triangulation overlay is not unique. Here, the DT protocol forces one of the nodes to change its coordinates by a small amount. This ensures that the Delaunay triangulation of the nodes is unique.

Whenever a node receives a message from a node with the same coordinates, the receiver shifts its coordinates by a small amount. The receiver also removes all neighbors that fail the neighbor test with the new coordinates. If a node A receives a message from a node B and a node in A's neighborhood table has the same coordinates as B, then A will send a *HelloNotNeighbor* message to B. Since the *HelloNotNeighbor* message contains A's neighbor with B's logical address, B sends the node with the duplicate logical address a *HelloNeighbor* message. The receiver of this *HelloNeighbor* message notices that the message was sent by a node with the same coordinates and changes its logical address.

If a node A receives a *HelloNeighbor* or *HelloNotNeighbor* message from a node N such that the sender N, the receiver A, the CW and CCW neighbors of A with respect to N, $CW_N(A)$ and C $CW_N(A)$, contained in the message lie on a circle, then A will shift its coordinates before processing the message.

Each time a node receives a *HelloNeighbor* or *HelloNotNeighbor* message from a neighbor, it checks whether or not the neighbor's logical address has changed. If the logical address has changed, then the node removes the neighbor's entry from its neighborhood table and then processes the message. In most cases, the node with the shifted coordinates will be added again as a neighbor.

3.3.11 States and State Transitions of the DT Protocol

We next discuss the states and state transitions of the DT protocol. The discussion summarizes our earlier description of the protocol. The DT protocol has two different finite state machines, one for a node and one of the DT server. A detailed description of the state transitions will be presented in tabular form in this section.

3.3.11.1 Node States

The state of a node is derived from the neighborhood table and the presence of candidate neighbors. There are no variables that memorize the states of a nodes. A node is in one of five states: *Stopped*, *Leader without Neighbor*, *Leader with Neighbor*, *Not Leader*, and *Leaving*. Recall that a node is a Leader if the node that has no neighbor with greater coordinates than its own. By definition, a node with no neighbors is a Leader. The states *Leader with Neighbor* and *Leade without Neighbor* are distinguished, for the following reason. When a newly joining node starts up or when a node has no neighbors, it believes itself to be a Leader, and it generate NewNode Messages. A node with neighbors does not send NewNode Messages. The definitions of the three states are given in Table 16.

State Name	State Definition
Stopped	The node is not running
Leaving	The node is going to leave the group
Leader without Neighbor	The node that has no neighbors
Leader With Neighbor	The node that has neighbors, and no neighbor has greater coordinates than its own
Not Leader	The node has a neighbor with coordinates greater than its own

Table 3.16**. Node State Definitions.**

For nodes in states *Leader with Neighbor* and *Not Leader*, we define three sub-states: *Stable With Candidate Neighbor*, *Stable Without Candidate Neighbor*, and *Not Stable*. We say a node X is stable when all nodes that appear in the CW and CCW neighbor columns of node X’s neighborhood table also appear in the neighbor column. We say a node M has a candidate neighbor, say node N, if (1) N appears in the CW or CCW column of M’s neighborhood table, or it is contained in a NewNode message received by M, and (2) N is not in the neighbor column of M’s neighborhood table, and (3) N passes the neighbor test at M. The definitions

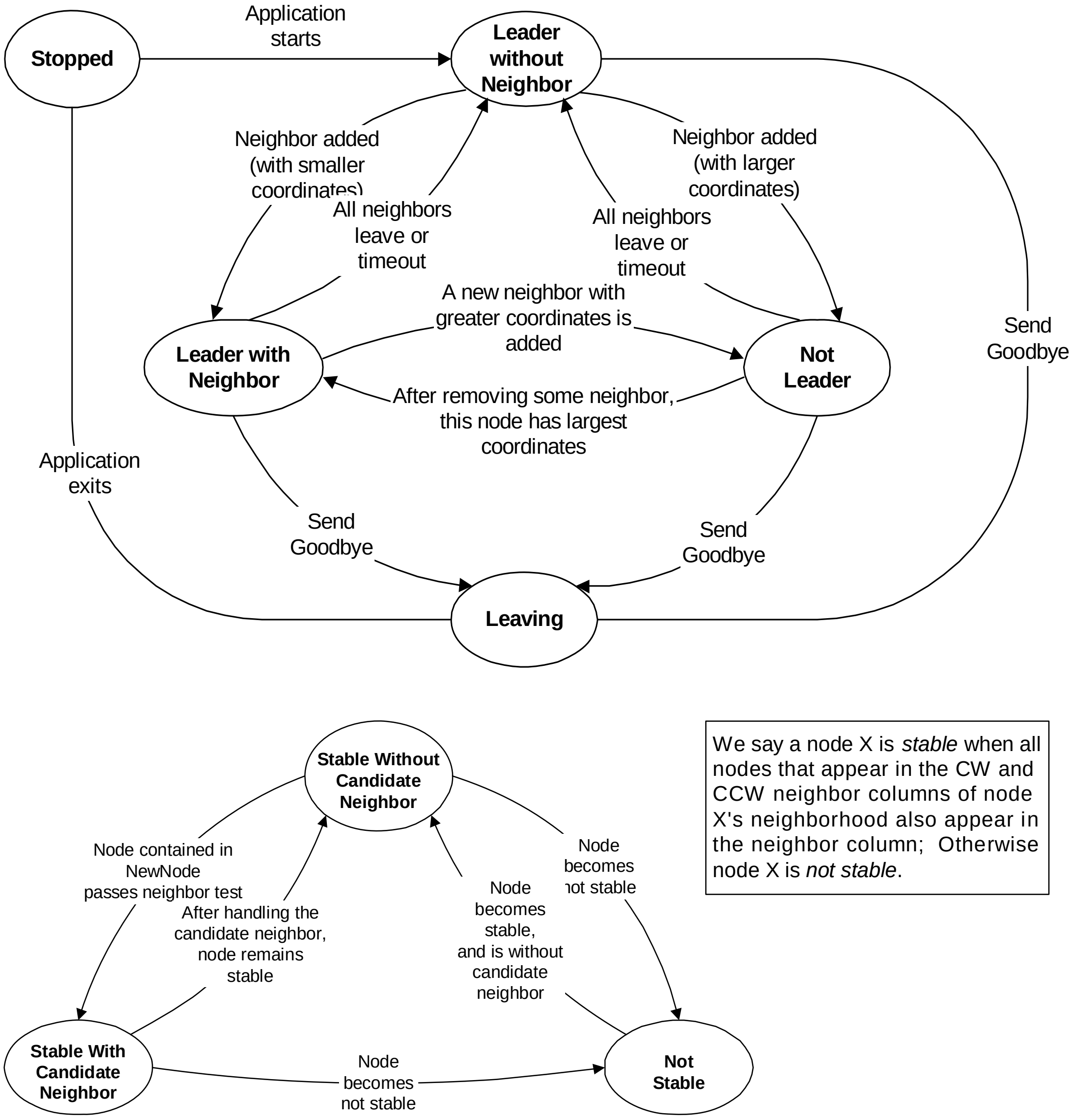
of the three sub-states are given in Table 3.17.

Sub-state Name	State Definition
Stable Without Candidate	The node is stable and has no candidate neighbors

Neighbor	
Stable With Candidate Neighbor	The node is stable and has candidate neighbors
Not Stable	The node is not stable

Table 3.17**.**Node Sub-state Definitions.

A new node starts in state **Stopped**. When it is in state **With Neighbor Leader** and **Not Leader**, the node also has a sub-state. The transition diagram of states and sub-states is shown in Figure 17.


Stopped
Application starts
Leader without Neighbor
Neighbor added (with smaller coordinates)
All neighbors leave or timeout
Neighbor added (with larger coordinates)
All neighbors leave or timeout
A new neighbor with greater coordinates is added
Leader with Neighbor
Not Leader
After removing some neighbor, this node has largest coordinates
Send Goodbye
Application exits
Send Goodbye
Send Goodbye
Leaving
Stable Without Candidate Neighbor
Node contained in NewNode passes neighbor test
After handling the candidate neighbor, node remains stable
Node becomes stable, and is without candidate neighbor
Node becomes not stable
Stable With Candidate Neighbor
Node becomes not stable
Not Stable

We say a node X is *stable* when all nodes that appear in the CW and CCW neighbor columns of node X's neighborhood also appear in the neighbor column; Otherwise node X is *not stable*.

Figure 3.17. **State and Sub-state transition diagrams**

3.3.11.2 DT Server States

The functions performed by the DT server are minimal. It is used as rendezvous point when new nodes join the overlay network and when the overlay Newark must be repaired after a partition. The DT server has only two states: *Has Leader* and *Without Leader*. Recall that the DT server maintains a cache of nodes. The node with the highest logical address is identified by the DT server as the Leader of the overlay network. If the node cache is empty, the DT server has no information about nodes in the overlay network. This state is referred to as *Without Leader*. If the node cache is not empty, the DT server can identify the Leader of the overlay network. This state is referred to as *Has Leader*. The definitions of the two states are given in Table 3.18.

State Name	State Definition
Has Leader	The node cache contains at least one node
Without Leader	The node cache is empty

Table 3.18**: DT Server State Definitions.**

The state transition diagram of the DT server is shown in Figure 3.18. The DT Server starts in state **Without Leade**r. When the first joining node sends a ServerRequest message to the DT server, this node is added to the node cache, and the DT server will enter state **Has Leade**r.

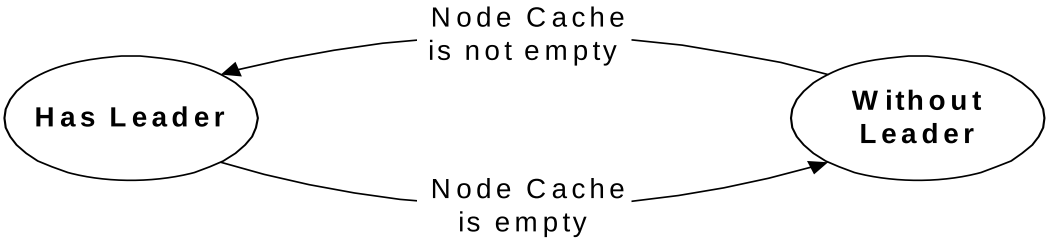
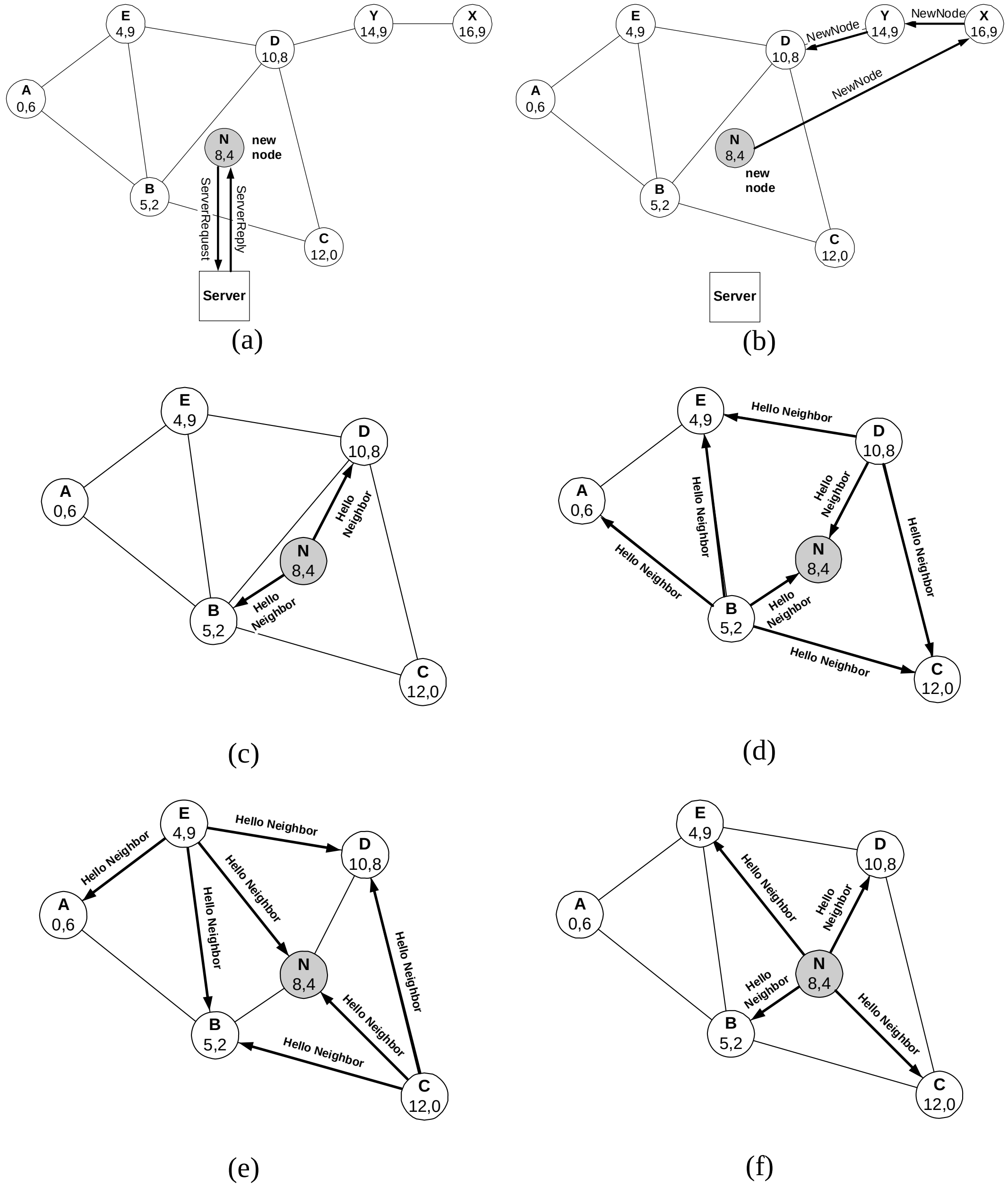


Figure 3.18. **DT Server State Transition Diagram.**

3.3.12 Examples

In an overlay network that has been stable for an extended period of time, there are three types of events: (1) all nodes send *HelloNeighbor* messages to their neighbors every $t_{SlowHeartbeat}$ seconds; (2) the Leader exchanges *ServerRequest* and *ServerReply* messages with the DT server every $t_{FastHeartbeat}$ seconds; (3) the server exchanges CachePing and CachePong messages with the nodes in its cache every $t_{SlowHeartbeat}$ seconds.

In the following diagrams, we illustrate the dynamics of the DT protocol. We illustrate the case when a node joins and the case when a node leaves the Delaunay triangulation.



(a) (b)

(c) (d)

(e) (f)

Figure 3.19. Joining node. Node N with coordinates coord(N)=(8,4) joins the overlay network. Note that in (a) and (b), we have omitted some edges from the Delaunay triangulation for the sake of simplicity. Also, nodes X and Y are omitted in (c)-(f).

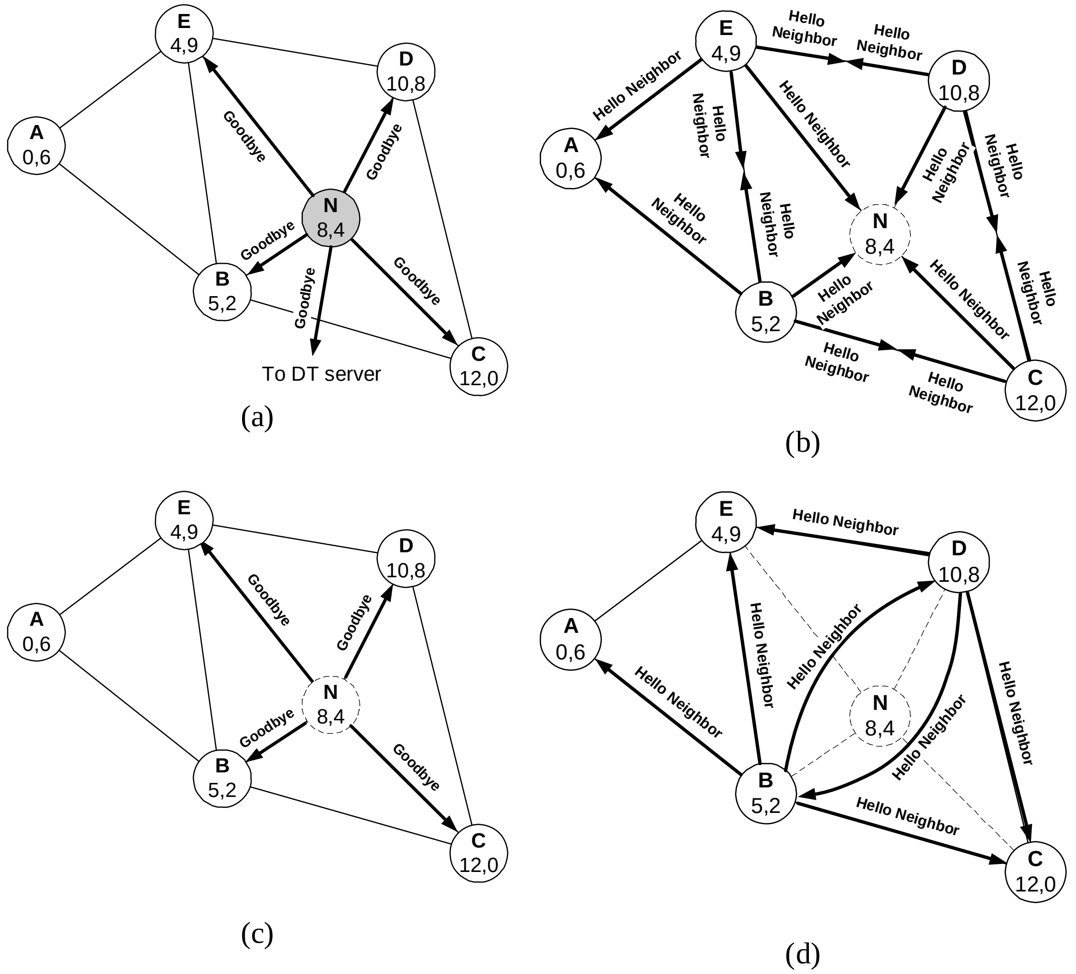


Figure 3.20. Node N with coordinates coord(N)=(8,4) leaves the overlay network.

In Figure 3.19 illustrate the steps of the DT protocol when a new node N with coord(N)=(8,4) joins an overlay network. As shown in Figure 3.19(a), N first sends a *ServerRequest* to the DT server. Then, N receives a ServerReply which contains the logical and physical addresses of some node X with coord(X) > coord(N). Next, N sends a *NewNode* message to X (Figure 3.19(b)). X performs a neighbor test for N, which fails. Therefore, X forwards the *NewNode* message to a neighbor Y, which is closer to N than X. Assuming that N fails the neighbor test at Y, Y forwards the *NewNode* message to D, which is closer to N than Y. At node D, N passes the neighbor test. Therefore, D makes N a candidate neighbor and sends a *HelloNeighbor* message to N. Thus, N has found its first neighbor.

Since the *HelloNeighbor* from D in Figure 3.19(b) contains B = CW_N(D) and C= CCW_N (D), nodes B and C become candidate neighbors at N. At the next timeout of the Heartbeat timer, N sends a *HelloNeighbor* to its neighbor D and its closest candidate neighbor B (Figure 3.19(c)). As soon as these *HelloNeighbor* messages are received at B and D, these nodes will drop each other from their neighborhood tables. In other words, the link in the overlay between nodes B and D will be removed.

In Figure 3.19(d), we assume that the Heartbeat timer expires at both B and D. Note that the sequence of events in this example is different if the Heartbeat timers expire in a different order. The nodes send *HelloNeighbor* messages to all their neighbors. When N receives the message from B, it promotes B from a candidate neighbor to a neighbor. The messages from B to E and from D to C contain N as a CW or CCW neighbor. Hence, N becomes a candidate neighbor at both C and E.

Assuming that the next Heartbeat timeout occurs at nodes C and E, these nodes send *HelloNeighbor* messages to all their neighbors and to their candidate neighbor N (Figure 3.19 (e)). When N receives the messages from C and E, it adds these nodes as neighbors. Now, N has a correct view of its neighborhood.

At the next Heartbeat timeout at N, shown in Figure 3.19(f), N sends *HelloNeighbor* messages to nodes B, C, D, and E. When the respective *HelloNeighbor* messages arrive at C and E, these nodes promote node N from candidate neighbor to neighbor. This completes the procedure for adding node N to the overlay network. Subsequently, each node sends *HelloNeighbor* messages to its neighbors at each Heartbeat timeout.

3.1.23 When a Node Leaves the Overlay

In Figure 3.20 we illustrate the steps involved in node N leaving the overlay. When N decides to leave the overlay, it sends *Goodbye* messages to all of its neighbors and to the DT server (Figure 3.20(a)). When the server receives the *Goodbye* message, it removes N from the cache. When N's neighbors receive the *Goodbye* message, they remove N from their neighborhood tables.

Although N is deleted from the neighborhood tables of nodes B, C, D, and E, these nodes have other neighbor entries where N is listed as a CW or CCW neighbor. For example, since N= CW_E(B) and N = CCW_E(D), node N appears as CW neighbor of B and as CCW neighbor of D in E's neighborhood table. Therefore, N is now a candidate neighbor at all of these nodes. Thus, these nodes will send *HelloNeighbor* messages to N at the Heartbeat timeout.

Let us now assume that all nodes send *HelloNeighbor* messages to their neighbors and their candidate neighbor N (Figure 3.20(b)). When N receives the messages, it will respond with *Goodbye* messages, as shown in Figure 20(c). The *HelloNeighbor* messages that are sent in Figure 3.20(b) contain the updated values of the CW and CCW neighbors of the nodes. For instance, B's message to E lists C (and no longer N) as the CW neighbor of B with respect to E, so C= CW_E(B). As a result, after Figure 3.20(b), node N no longer exists as a CW or CCW neighbor in the neighborhood tables of any node. Furthermore, nodes B, C, D, and E, know about each other either as neighbors or as a CW or CCW neighbor of some neighborhood table entry. When the neighbor tests are executed, C will fail the neighbor tests at node E and vice versa.

On the other hand, D passes the neighbor test at node B and B passes the test at node D. Hence, nodes B and D add each other as candidate neighbors and send HelloNeighbor messages to each other (Figure 3.20(d)). Once these messages are received, both B and D have established each other as neighbors and as a result the overlay network has been repaired.

Evaluation

We have evaluated the performance characteristics of the DT protocol in measurement experiments on a cluster of Linux PCs. The experiments included up to 100 PCs and overlay networks with up to 10,000 nodes. The measurement experiments are reported in [LIEBE01b].

We do not have a formal proof that the protocol always generates a Delaunay triangulation. However, we have verified that the network topologies that are generated by the protocol are Delaunay triangulations.

References

[BEAM99] T. K. Beam. HyperCast:A Protocol for Maintaining a Logical Hypercube-Based Network Topolog. M.S. Thesis, University of Virginia, May 1999.

[HOLZ97] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering.* Vol. 23, No. 5, May 1997.

[KRANA99] E. Kranakis, H. Singh, and J.Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry (CCCG'99),* 1999.

[LIEBE98a] J. Liebeherr and B. S. Sethi. "*Towards Super-Scalable Multicast*". Technical Report, Polytechnic University, CATT 98-121. January 1998.

[LIEBE98b] J. Liebeherr and B. S. Sethi. "*A Scalable Control Topology for Multicast Communications*". Proceedings of IEEE Infocom 1998.

[LIEBE99] J. Liebeherr and T. K. Beam. HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Proceedings First International Workshop on Networked Group Communication (NGC '99)*, Lecture Notes in Computer Science, Volume 1736, pages 72-89, 1999.

[LIEBE01a]] J. Liebeherr and M. Nahas. Application-layer Multicast with Delaunay Triangulations. *Proceedings of IEEE Globecom 2001*, Global Internet Symposium.

[LIEBE01b] J. Liebeherr, M. Nahas, and W. Si. Large-scale Application-Layer Multicast with Delaunay Triangulations. Manuscript, September 2001.

[LORIN01] K. Lorintz, HyperCast: A Super-Scalable Many-to-Many Multicast Protocol for Distributed Internet Applications, Undergraduate Thesis, School of Engineering and Applied Science, University of Virginia. May 2001.

[SIBS77] R. Sibson. Locally equiangular triangulations. *The Computer Journal*, 21(3):243--245, 1977.

Appendix A: Actions of the DT protocol

The following tables show the actions taken by the nodes in each state when events like message arrivals, timer expirations happen. We do not have separate tables for the three sub-states; the description of the actions in the sub-states are included in the **With Neighbor Leader** and **Not Leader** tables. We use the following notations and terminology:

ServerReply(*x*):	Indicates a ServerReply message which contains node *x*.
NewNode(*w*):	Indicates New Node message which contains node *w*.
→ :	Indicates a state transition.
return :	Processing for this event is complete. Skip the remainder.
this :	Refers to the local node.

A.1 Transition Table for Node

The following are transitions at node *v*.

State: **Stopped**

Event	Action
Application starts	→ **Leader Without Neigbhor**

States: **Leader with Neighbor, Leadwer without Neighbor, Not Leader**

Event	Action
Application exits	Send Goodbye to all neighbors Send Goodbye to server Ã→ **Leaving**
CachePing received	Reply with CachePong message
NewNode(*w*) received	IF *w* passes neighbor test at *v* /* *w* is a candidate neighbor */ Send HelloNeighbor to *w* Set timeout value of Heartbeat Timer to Fastheartbeat ELSE Forward message to a neighbor which is closer to *w*

State: **Leader With Neighbor, Not Leader**

Event	Action
Heartbeat Timer expires	Send HelloNeighbor to all neighbors IF node is not stable Send HelloNeighbor to closest candidate neighbor Set timeout value of Heartbeat Timer to Fastheartbeat ELSE Set timeout value of Heartbeat Timer to Slowheartbeat

State: **Leader Without Neighbor**

Event	Action
Backoff Timer expires	IF $t_{Backoff} \geq t_{Neighbor}$ and an alternate DT server exists Switch to alternate DT server Set $t_{Backoff}$ to $t_{FastHeartbeat}$ Send ServerRequest to alternate DT server ELSE Send ServerRequest to DT server Set $t_{Backoff} = 2t_{Backoff}$ Start Backoff Timer
Receives ServerReply(*w*)	Set $t_{Backoff} = t_{FastHeartbeat}$ IF $w \neq this$ Send NewNode(*this*) to w.
HelloNeighbor arrives from node *w*	While $Coord_{this} = Coord_{w}$ Node shift its coordinates IF *w* passes neighbor test Add *w* as neighbor in neighborhood table Start Neighbor Timer for *w* with $t_{Neighbor}$ IF $Coord_{w} > Coord_{this}$ → **Not Leader** Else Set $t_{Backoff} = t_{FastHeartbeat}$ → **Leader with Neighbor**

State: **Leader With Neighbor**

Event	Action
Backoff Timer expires	IF $t_{Backoff} \geq t_{Neighbor}$ and an alternate DT server exists Switch to alternate DT server Set $t_{Backoff}$ to $t_{FastHeartbeat}$ Send ServerRequest to alternate DT server ELSE Send ServerRequest to DT server Set $t_{Backoff} = 2t_{Backoff}$ Start Backoff Timer
Receives ServerReply(*w*)	Set $t_{Backoff} = t_{FastHeartbeat}$ IF $w \neq this$ Send NewNode(*this*) to w.
Neighbor Timer for *w* expires or Goodbye arrives from *w*	IF *w* is neighbor Remove *w* from neighborhood table Set timeout value of Heartbeat Timer to Fastheartbeat IF no neighbors in neighborhood table → **Leader without neighbors**
HelloNeighbor or HelloNotNeighbor arrives from node *w*	Ã IF *w* is a neighbor IF *w* has changed its coordinates Remove *w* from neighbor table **Return** Update neighborhood entry for *w* Start Neighbor Timer for *w* IF node is stable Set Heartbeat Timer to Slowheartbeat Else Set Heartbeat Timer to Fastheartbeat Else /* *w* is not a neighbor */ IF another neighbor $v \neq w$ exists such that $Coord_v = Coord_w$ Send HelloNotNeighbor to w Else While $Coord_{this} = Coord_w$, or for any neighbor v,

	$Coord_{this} = Coord_v$ Node shift its coordinates IF *w* passes neighbor test Add *w* as neighbor in neighborhood table Set Neighbor Timer for *w* Remove all neighbors that fail neighbor test While there are four nodes with coordinates on a circle Node shifts its coordinates While there exists neighbor v with $Coord_{this} = Coord_v$ Node shifts its coordinates Remove all neighbors that fail neighbor test Else if message is HelloNeighbor and *w* fails neighbor test Send HelloNotNeighbor to *w* IF there exists neighbor *v* with $Coord_v > Coord_{this}$ Clear the Backoff Timer(if the timer was set) → **Not Leader**

State: **Not Leader**

Event	Action
Backoff Timer expires	IF $t_{Backoff} \geq t_{Neighbor}$ and an alternate DT server exists Switch to alternate DT server Set $t_{Backoff}$ to $t_{FastHeartbeat}$ Send ServerRequest to alternate DT server Else Send ServerRequest to DT server Set $t_{Backoff} = 2t_{Backoff}$ Start Backoff Timer
Receives ServerReply(*w*)	Set $t_{Backoff} = t_{FastHeartbeat}$ IF $w \neq this$ Send NewNode(*this*) to w.
Neighbor Timer for *w*	IF *w* is neighbor

expires or Goodbye arrives from *w*	Remove *w* from neighborhood table Set timeout value of Heartbeat Timer to Fastheartbeat IF no neighbors in neighborhood table → **Leader without neighbors**
HelloNeighbor or HelloNotNeighbor arrives from node *w*	Ã IF *w* is a neighbor IF *w* has changed its coordinates Remove *w* from neighbor table **Return** Update neighborhood entry for *w* Start Neighbor Timer for *w* IF node is stable Set Heartbeat Timer to Slowheartbeat Else Set Heartbeat Timer to Fastheartbeat Else /* *w* is not a neighbor */ IF another neighbor $v \neq w$ exists such that $Coord_v = Coord_w$ Send HelloNotNeighbor to w Else While $Coord_{this} = Coord_w$, or for any neighbor v, $Coord_{this} = Coord_v$ Node shift its coordinates IF *w* passes neighbor test Add *w* as neighbor in neighborhood table Set Neighbor Timer for *w* Remove all neighbors that fail neighbor test While there are four nodes with coordinates on a circle Node shifts its coordinates While there exists neighbor v with $Coord_{this} = Coord_v$ Node shifts its coordinates Remove all neighbors that fail neighbor test Else if message is HelloNeighbor and *w* fails neighbor test

	Send HelloNotNeighbor to *w*

State: **Leaving**

Event	Action
Application exits	→ **Stopped**
Goodbye arrives from *w*	Ã Do nothing.
A message (not Goodbye arrives from *w*)	Send Goodbye to *w*

A.2 Transition Table for DT Server

The actions of the server in its two states are shown in the following two tables.

State: **Without Leader**

Event	Action
Server receives ServerRequest from *v*	Add node *v* to node cache Start Cache Timer for *v* Set Leader:=*v* ä Start Leader Timer Send ServerReply(*v*) to node *v* → **Has Leader**

State: **Has Leader**

Event	Action
Server receives ServerRequest from *v*	IF Leader = *v* or *v* is in node cache IF v has changed its coordinates Update *v*'s stored coordinates ELSE IF $Coord_v > Coord_{Leader}$ Set Leader := *v* Start Leader Timer

	IF node cache is full Remove one node cache entry Add node *v* to node cache ELSE IF node cache is not full Add *v* to node cache Start Cache Entry Timer IF Leader = *v* Send ServerReply(*v*) to node *v* ELSE Selcet *w* from node cache with $Coord_w > Coord_v$ Send ServerReply(*w*) to *v* IF Leader != *w* and *w* has been used in 6 ServerReplys Remove w from cache
Goodbye received from *v* or Cache Entry Timer for *v* expires or Leader Timer for *v* expires	IF v is in node cache Remove *v* from node cache IF Leader = *v* and cache is not empty Select new Leader = *y*, where *y* is the node in the node cache with largest coordinates ELSE IF Leader = *v* and cache is empty → Without Leader
Heartbeat Timer expires	Ã Send CachePing messages to every node in node cache
CachePong received From *v*	Restart Cache Entry timer for *v*

Appendix B. DT Protocol Message Format

All DT protocol messages have the same format with the same set of fields. However, the same fields may be interpreted differently dependent on the message type. The message format is shown in Figure 21.

1 byte	4 bytes	14 bytes	14 bytes	14 bytes	14 bytes
Type	OverlayID Hash	SRC	DST	ADDR1	ADDR2

Figure 21. Message format of DT protocol

The type of the DT protocol message is indicated by a 1-byte long Type field.

Message Type	**Type Field**
HelloNeighbor	0
HelloNotNeighbor	1
Goodbye	2
ServerRequest	3
ServerReply	4
NewNode	5
CachePing	6
CachePong	7

The OverlayIDHash is a 4-byte long hash value which is derived from the OverlayID. If the OverlayID is composed of only ASCII characters, we apply the hash function to the byte array of these ASCII characters. If the OverlayID contains non-ASCII characters, we require that the character encoding scheme is UTF-8, then we apply the hash function to the raw byte array of the UTF-8 encoding. The hash function, which can operate on variable-length byte arrays, is as follows:

Input: byte array, denoted by “A[ ]”

Output: a 4-byte unsigned integer, denoted as “result”

Operators:

Op1 >> Op2: “Op1” is bit-wise right shifted “Op2” times.

Op1 << Op2: “Op1” is bit-wise left shifted “Op2” times.

Op1 & Op2: bit-wise AND of “Op1” and “Op2”.

```
        Op1  ^  Op2:   bit-wise XOR of “Op1” and “Op2”.
Procedure OverlayIDHash (byte  A[] )
Begin
   Result := 0;
   For ( int i := 0 ; i < length of A[ ] ; i++ ) {
         Byte  upperByte := (byte) ( (result >> 24) & 0xFF );
         int    leftShiftValue := ((upperByte ^ A[i]) & 0x07) + 1;
         result := ((result << leftShiftValue) ^ ((upperByte ^ A[i]) & 0xFF));
   }
   return result;
end
```

Table 3.19 The SRC, DST, ADDR1, and ADDR2 fields each contain the logical and physical addresses of a node.

A logical address consists of the (x,y) coordinates of the Delaunay triangulation, where x and y are each a 4-byte unsigned integer. A physical address consists of an IP address and a port number, where the IP address is 4 bytes long and the port number is 2 bytes long. So, the entire length of an address field with a logical and a physical address is 14 bytes. The exact format is shown in Figure22.

4 bytes	4 bytes	4 bytes	2 bytes
x-coordinate of logical address	**y-coordinate of logical address**	**IP address**	**port number**

Figure 22. Format of a logical address/physical address.

- **HelloNeighbor/HelloNotNeighbor:** SRC and DST contain the addresses of the sending and receiving node. ADRR1 and ADDR2, respectively, are the address of the CW and CCW neighbors of the sender with respect to the destination. If the sender has no CW or CCW neighbors, the corresponding fields are set to zero.
- **Goodbye:** If the message is sent to the DT server, DST is set to all zeros. Otherwise, DST contains the address of the receiving node. The fields ADDR1 and ADDR2 are set to zero.
- **ServerRequest:** SRC contains the address of the sending node. The fields DST, ADDR1, and ADDR2 are set to zero.
- **ServerReply:** The IP address and port number portion of the SRC field are set to the IP address and the port number of the DT server. The logical address part of field SRC is set to zero (Note that the DT server does not have a logical address). DST is the address of the node that sent the corresponding ServerRequest. The field ADDR1 has the address of a node with a larger logical address (coordinates) than the logical address (coordinates) in the DST field. If

the DT server does not know about a node with a larger logical address, i.e., the DT server believes that the node described in the DST field is a Leader, then the ADDR1 field is set to be equal to the DST field. The field ADDR2 is set to zero.

- **NewNode:** The fields SRC and DST contain the sender and receiver, respectively, of the message. Whenever, the NewNode message is forwarded to another node, the fields SRC and DST are updated. ADDR1 contains the node who initially sends the NewNode message, i.e., the "new node". ADDR2 is set to zero.
- **CachePing:** The SRC contains the IP and port number of the DT server, with the logical address part of the address set to zero. The DST field contains the address of the receiving node. The ADDR1 and ADDR2 fields are set to zero.
- **CachePong:** SRC contains the address of the sending node. DST is IP address and port number of the DT server, as contained in the CachePing message. The fields ADDR1 and ADDR2 are set to zero.

DT protocol versions

> The following is an update of on the HyperCast protocol changes from 2004.
>
> The Landmark option was implemented, but has been deleted.

The DT protocol uses (x,y) coordinates (where x,y are integer numbers) as logical addresses. There exist multiple DT protocol versions due to the usage of different methods through which the logical address can be assigned, and rendezvous mechanisms. This document describes all of the existing variants of the DT protocol.

In HCAST2.0 (HyperCast Version 2), for coordinate assignment, three methods are available: static assignment, coordinates derived from IP address, and random assignment. The methods are determined by selecting the attribute DT2-0.Coords.

x,y: specify the coordinates of the node.
USE_IP: coordinates of the node are derived from IP address.
RANDOMxxxx (xxxx is a number such as 1000): coordinates of the node are randomly selected from a xxxx by xxxx grid.

HC3.0 (HyperCast Version 3.0) provides two additional methods to assign coordinates:

USE_LM: coordinates of the node are determined by measuring the delays to a predefined set of nodes called Landmarks.

USE_GEO: coordinates of the node are derived from longitude and latitude.

3.1.24 Rendezvous methods

Generally, a new node can use one of the following methods to rendezvous with an existing DT overlay: a dedicated server (rendezvous server), a list of potential members (buddylist) and announcement via multicast or local broadcast (broadcast). In HCAST2.0, only the rendezvous server is supported. In HCAST3.0, all three rendezvous methods are supported.

- **Rendezvous server**

With the server rendezvous, a dedicated server which is not part of the overlay is employed for node joining objective. The rendezvous server caches a list of existing overlay nodes. When a new node joins the overlay, it first sends a request to the rendezvous server, which provides information to contact with the overlay.

- **Buddylist**

With a buddylist rendezvous, a newly joining node has (by configuration) a list of possible members, called the buddylist. The new node joins the overlay by trying to contact the nodes in the buddylist. The node in the buddylist provides information for contacting the overlay.

- **Broadcast**

A broadcast rendezvous server assumes the existence of a broadcast channel, e.g., IP multicast or local broadcast. A new node sends join request to and receives reply from this channel. By this way, the new node can establish the connection with the existing nodes, and the existing overlay. In HCAST3.0, IP multicast is used to provide the functions of the broadcast channel.

3.1.25 Coordinate selection with LANDMARKS

Design Overview

The method for determining a node's coordinates by employing Landmarks is based on the work of Global Network Positioning (GNP) by Eugene Ng. GNP models the Internet as a Euclidean coordinate system. A number of landmark hosts are placed on the Internet as the reference points. The operations can be divided into two phases:

- First, these landmarks measure the delays between each other and determine their coordinates by solving an optimization problem that minimizes the difference between measured and estimated delays.
- Second, after the coordinates of the landmarks are determined, the coordinates and landmarks' addresses information (IP address & port number) are published to a well-known website or an overlay management server (in the case of Hypercast), so that the information can be widely accessed. In the second phase, a node who needs to determine its logical coordinates makes delay measurements to the landmarks and calculates its coordinates via a triangulation.

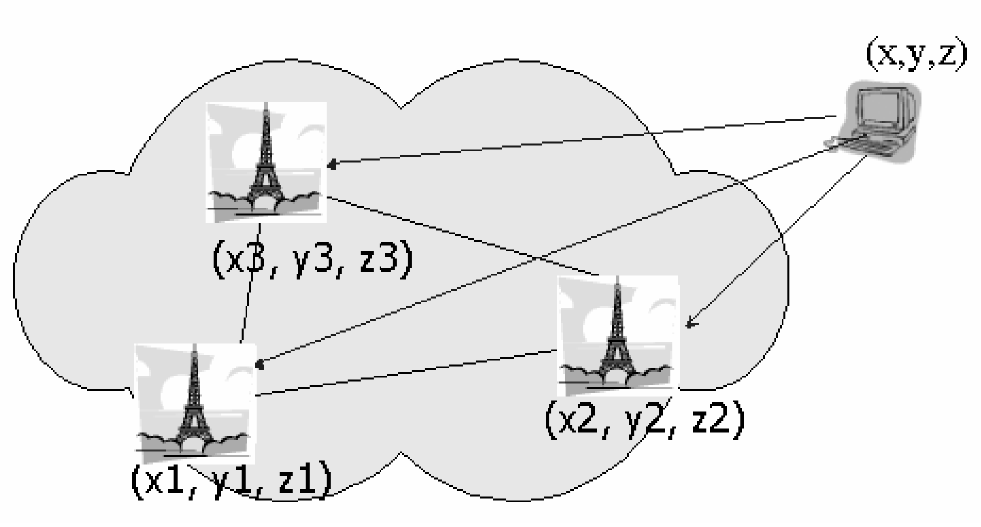


Figure 1. GNP system

As shown in Figure 1, the two key parts are the landmarks and the targets, the hosts that need logical coordinates. In the implementation, the landmark is implemented as a daemon process running on a number of pre-selected long-live hosts. The other alternative to this daemon process is to probe the well-known port of the landmarks. This can simplify the whole design because the design and implementation of the daemon process can be saved. However, this approach has two disadvantages. First, not all the hosts support those well-known ports and actually many publicly accessible hosts disable those ports for security reasons. Second, to obtain all the pair-wise delays between all the landmarks, it is inevitable to obtain accounts of those landmark hosts. Therefore, the complexity and cost of this approach cannot be lowered considerably, compared to the daemon process approach. The target is implemented as a configurable and pluggable component so that it can be seamlessly integrated into the existing Hypercast software. The communications between landmarks themselves and between landmarks and targets use message format explained in details in section 3.

To make the configuration of GNP consistent with the current Hypercast configuration, we extend the configuration file and ensure modifications are backward compatible. Below lists the Landmark related properties for single node case. The lines in the configuration file are shown in dark-blue bold font and the explanation is shown in Italic font.

DT3-0.Coords = USE_LM

To use GNP to determine an overlay node's logical coordinates, one more option for the attribute DT3-0.Coords is added.

DT3-0.LandmarkNum = 3

Indicate the number of landmarks used in an overlay

DT3-0.Landmark0 = 128.143.69.149:3456

The physical address (IP+UDP port number) of the first landmark

DT3-0.Landmark1 = 128.143.137.16:3456

The physical address (IP+UDP port number) of the second landmark

DT3-0.Landmark2 = 128.143.137.15:3456

The physical address (IP+UDP port number) of the third landmark

DT3-0.Landmark0.Coordinates = 108.37761,102.059715

The logical coordinates of the first landmark

DT3-0.Landmark1.Coordinates = 108.91815,106.42619

The logical coordinates of the second landmark

DT3-0.Landmark2.Coordinates = 104.11766,102.20526

The logical coordinates of the third landmark

Software Architecture, in OL Socket & LM Design

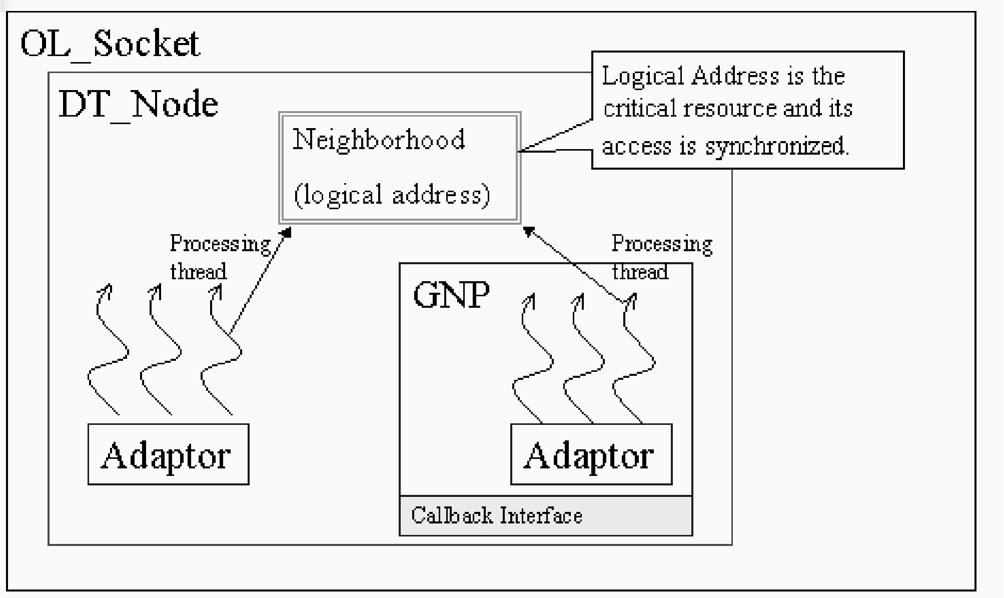


Figure 2. GNP component in OL_Socket

As the figure shows, GNP component (target) is a configurable part inside DT_Node. Three options are considered during the design phrase. First, GNP component can be placed outside the overlay node process. In that case, GNP component acts as a location service running on the local machine. The main advantage of the first choice is that it ensures all the overlay nodes on the same machine can obtain the same logical address and consequently, leads to a host-specific optimization for the overlay topology. For example, only one overlay node on the local host is elected to communicate with outside. This can eliminate the many duplicated logical links originated from the same node. However, the major concern for this choice is the high cost of inter-process communication, complicated dependency between processes and difficulty of updating logical addresses. The second choice is to place GNP component inside OL_Socket, but outside DT_Node. There are two main disadvantages of doing that. One is the introduction of inconsistency in software architecture. Adding GNP component to OL_Socket means that this component is a member of OL_Socket. However, not every overlay node needs such information, for example, Hypercube. Such a design does not abstract the essential common functions well. The other disadvantage is that the communication between GNP component and overlay node, is still complicated. Therefore, we choose the third option of placing GNP component inside DT Node. In this way, the whole software architecture is keep intact and all the changes are limited to DT node. In addition, since GNP component is a member of DT node, the communication between the two is simple and easy to implement.

In the GNP component, the UDP_UnicastAdaptor is used to for the probing of landmarks. GNP component implements the I_AdapterCallBack interface to process the messages to and from the landmarks. Logical address is a critical resource because GNP component's processing thread and DT node's process threads can access it concurrently. The logical address of the DT node is stored at the member neighborhood. To ensure the consistence of the data, Java synchronization is used to protect the critical resource. The other part of the system is landmark daemon. The landmark is also implemented using UDP_UnicastAdapter.

3.1.26 Coordinate selection USING Geographical Coordinates

Algorithm

Geographical coordinates are two-dimensional coordinates obtained from the longitudes and latitudes of a node. For a longitude-latitude pair (lo, la), we know the range of their values:

(south) -90 <= la <= +90 (north)

(west) -180 <= lo <= +180 (east)

Given a set of points on the earth, each of them corresponds to a longitude-latitude pair (x, y), we can draw a rectangle area which contains all points. To ensure the DT formed by this set of points resides in this rectangle area, we choose a base meridian, which is outside of the rectangle area, as the reference line. The value of the base meridian is denoted by base_meridian. The objective of this algorithm is to calculate the two-dimensional coordinates (x_cor, y_cor) from the value of (x, y) and base_meridian. For each point, its x-coordinate is calculated based on the adjusted difference between the longitude of this point and the base meridian. If base meridian is not specified, the Prime Meridian will be used as the base meridian.

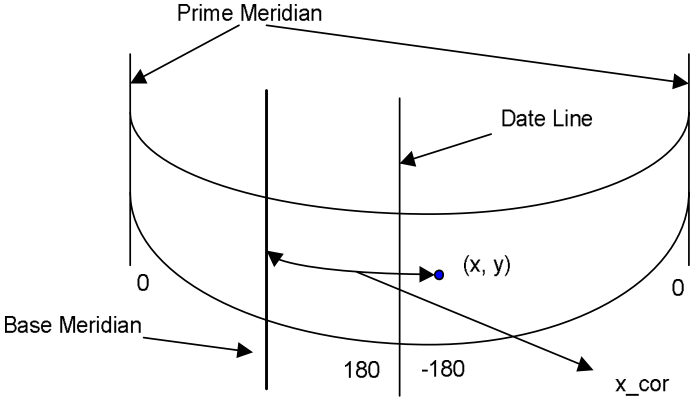


Figure 1. Base Meridian

Case 1: base_meridian >= 0

If x >= base_meridian, then x_cor = x - base_meridian;

If x < 0, then x_cor = (180 - base_meridian) + (x – (-180))

= 360 + x - base_meridian;

If 0<= x < base_meridian, then x_cor = 360 – (base_meridian – x)

= 360 + x - base_meridian.

Case 1: base_meridian < 0

If base_meridian <= x <= 0, then x_cor = x – base_meridian

If x > 0, then x_cor = (x – 0) + (0 – base_meridian) = x - base_meridian;

If x < base_meridian, then x_cor = 360 – (base_meridian - x)

= 360 + x - base_meridian;

Combine all cases, we can simplify the calculation:

If x >= base_meridian, then x_cor = x - base_meridian;

Else, x_cor = 360 + x - base_meridian.

For y_cor, the following calculation is used which ensures y_cor >= 0:

y_cor = y + 90

The coordinates calculated in the above way satisfy:

1. x_cor >= 0, y_cor >= 0;
2. the DT formed by the coordinates of the set of pints will not cross the base meridian line.

Implementation

To support the way through which a node derives its logical address from the longitude and latitude, the following properties are added in the Hypercast configuration file for single node case:

DT3-0.Coords = USE_GEO

To use geographical information to determine an overlay node's logical coordinate.

DT3-0.USE_GEO.GeoLocation = -95.2631, 38.9605

Indicate the longitude and latitude of the node.

DT3-0.USE_GEO.BaseMeridian = 0.0

Indicate the base or reference meridian line for the overlay.

In the DT_Node, DT_Node_Buddylist and DT_Node_Multicast class, the above algorithm is implemented in the method createLogicalAddress(). A modification is made to reduce the impact of the shift of a node's coordinates: after obtaining x_cor and y_cor, we set x_cor = x_cor * 10, and y_cor=y_cor*10.