Application-Layer Multicasting with Delaunay Triangulation Overlays*

Jörg Liebeherr *Michael Nahas* *Weisheng Si*

Department of Computer Science
University of Virginia
Charlottesville, VA 22904

Abstract

Application-layer multicast supports group applications without the need for a network-layer multicast protocol. Here, applications arrange themselves in a logical overlay network and transfer data within the overlay. In this paper, we present an application-layer multicast solution that uses a Delaunay triangulation as an overlay network topology. An advantage of using a Delaunay triangulation is that it allows each application to locally derive next-hop routing information without requiring a routing protocol in the overlay. A disadvantage of using a Delaunay triangulation is that the mapping of the overlay to the network topology at the network and data link layer may be suboptimal. We present a protocol, called *Delaunay Triangulation (DT)* protocol, which constructs Delaunay triangulation overlay networks. We present measurement experiments of the DT protocol for overlay networks with up to 10 000 members, that are running on a local PC cluster with 100 Linux PCs. The results show that the protocol stabilizes quickly, e.g., an overlay network with 10 000 nodes can be built in just over 30 seconds. The traffic measurements indicate that the average overhead of a node is only a few kilobits per second if the overlay network is in a steady state. Results of throughput experiments of multicast transmissions (using TCP unicast connections between neighbors in the overlay network) show an achievable throughput of approximately 15 Mbps in an overlay with 100 nodes and 2 Mbps in an overlay with 1 000 nodes.

Key Words: Application-Layer Multicasting, Multicasting, Group Communication, Delaunay Triangulation.

1 Introduction

Due to the lack of a widely available IP multicast service, recent research has examined implementing multicast services in the application layer. The general approach is to have applications self-organize into a logical overlay network, and transfer data along the edges of the overlay network using unicast transport services. Here, each application communicates only with its neighbors in the overlay network. Multicasting is implemented by forwarding messages along trees that are embedded in the virtual overlay network.

Application-layer multicast has several attractive features: (a) There is no requirement for multicast support in the layer-3 network; (b) There is no need to allocate a global group identifier, such as an IP multicast address; (c) Since data is sent via unicast, flow control, congestion control, and reliable delivery services that are available for unicast can be exploited. A drawbacks of application-layer multicast is that, since data is forwarded between end-systems, end-to-end latencies can be high. Another drawback is that,

*This work is supported in part by the National Science Foundation through grants NCR-9624106 (CAREER), ANI-9870336, and ANI-0085955.

if multiple edges of the overlay are mapped to the same network link, multiple copies of the same data may be transmitted over this link, resulting in an inefficient use of bandwidth. Thus, the relative increase of end-to-end latencies and the increase in bandwidth requirements as compared to network-layer multicast, are important performance measures for overlay network topologies for application-layer multicast.

Most overlay topologies for application-layer multicast fall into three groups. Topologies in the first group consist of a single tree [8, 9, 11, 13, 19, 22]. A drawback of using a single tree is that the failure of a single application may cause a partition of the overlay topology. The second group of topologies are mesh graphs, where data is transmitted along spanning trees which are embedded in the mesh graph [3, 4]. A drawback of mesh graphs is that the calculation of spanning trees requires running a multicast routing protocol (e.g., DVMRP) within the overlay, which adds complexity to the overlay network. Generally, in the aforementioned topologies, nodes in the overlay network send probe messages to each other to measure network-layer latencies. These measurements are used to build an overlay network that is a good fit for the network-layer topology. The third group of topologies assigns to members of the overlay network logical addresses from some abstract coordinate space, and builds the overlay network with the help of these logical addresses. For example, the overlay in [16] assigns each member of the overlay network a binary string, and builds an overlay network with a hypercube topology. In [21], logical addresses are obtained from n-dimensional Cartesian coordinates on an n-torus. An advantage of building overlay networks with logical addresses is that, for good choices of the address space and the topology, next-hop routing information for unicast and multicast transmissions can be encoded in the logical addresses. A disadvantage of building overlay networks using a logical address space is that the overlay network may not be a good match for the network-layer topology. Note that recent proposals for location services in peer-to-peer networks have adopted logical coordinate spaces that assign logical addresses to data items, e.g., [20, 24, 25]. Even though lookup services and application-layer multicast services have different objectives, some logical coordinate spaces can be applied in both contexts [21].

In this report we present an application-layer multicast solution that is suitable for very large group sizes with many thousand nodes. Our approach is to use a *Delaunay triangulation* as overlay network topology. The choice of the overlay topology falls into the third group of application-layer multicast solutions, which draw logical addresses from a coordinate space. Specifically, each member of an overlay network is assigned logical (x, y) coordinates in a plane. We show that Delaunay triangulations can be built in a distributed fashion, and that multicast trees can be embedded in a Delaunay triangulation overlay without requiring a routing protocol in the overlay. The mapping of a Delaunay triangulation overlay network to the network-layer infrastructure can be suboptimal, especially if the logical coordinates of a member in the overlay network are not well matched to the underlying network topology.

We present a protocol, called *DT Protocol*, which creates and maintains a Delaunay triangulation overlay of applications that are addressable on the Internet. The DT protocol achieves scalability through a distributed implementation where no entity maintains knowledge of the entire group. We evaluate the DT protocol through measurement experiments for overlay networks with up to 10 000 members on a cluster of PCs. The results show that the DT protocol can maintain a Delaunay triangulation overlay for a multicast group with dynamically changing group membership. In other experiments, we evaluated the performance of the overlay's topology using multicast batch file transfers to groups with up to 1 000 members.

The contribution of the presented Delaunay triangulation and the DT protocol is that we can build and maintain very large overlay networks with relatively low overhead, at the cost of suboptimal resource utilization due to a possibly poor match of the overlay network to the network-layer infrastructure. Hence, in terms of John Chuang's taxonomy of scalable services [5], the Delaunay triangulation overlay network trades-off economy-of-scale for increased scalability.

The remainder of this report is organized as follows. In Section 2, we define Delaunay triangulations and Delaunay triangulation overlay networks. In Section 3, we compare Delaunay triangulation overlay

networks with other proposed overlay network topologies. In Section 4, we describe the DT protocol. In Section 5, we present measurement experiments that evaluate the performance of our implementation of the DT protocol. In Section 6, we present brief conclusions.

2 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 1, we show a Delaunay triangulation and the circumscribing circles of some of its triangles. Delaunay triangulations have been studied extensively in computational geometry [7] and have been applied in many areas of science and engineering, including communication networks, e.g., [1, 12].

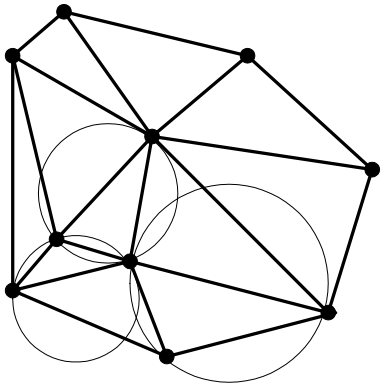


Figure 1: A Delaunay Triangulation.

2.1 Delaunay Triangulation Overlay Network

In order to establish a Delaunay triangulation overlay, each application, henceforth called '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 network, i.e., are *neighbors*, if their corresponding vertices are connected by an edge in the Delaunay triangulation that consists of all vertices associated with the nodes of the overlay.

The Delaunay triangulation has several properties that make it attractive as an overlay topology for application-layer multicast. First, Delaunay triangulations generally have a set of alternate non-overlapping routes between any pair of vertices. The existence of such alternate paths can be exploited by an application-layer overlay when nodes fail. Second, the number of edges at a vertex in a Delaunay triangulation is generally small. The average number of edges at each vertex is less than six. Even though, in the worst-case, the number of edges at a vertex is $n-1$, the maximum number of edges is usually small[1]. Third, once the topology is established, packet forwarding information is encoded in the coordinates of a node, without the 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.

2.2 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, and that has the sender as the root of the tree. In the

[1]The worst-case is created when $n-1$ vertices form a circle and the nth vertex is in the center of the circle.

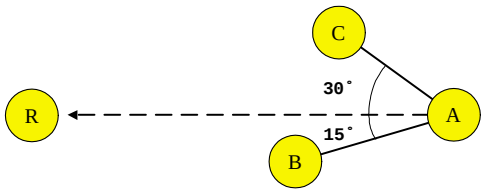


Figure 2: Compass Routing. Node A has two neighbors, B and C. A computes B as the parent in the tree with root R, since the angle $\angle RAB = 15°$ is smaller than the angle $\angle RAC = 30°$.

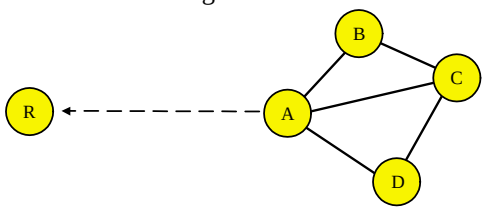


Figure 3: Compass Routing. Node A determines that it is the parent for node C, since the angle $\angle RCA$ is smaller than angles $\angle RCD$ and $\angle RCB$. Likewise, B and D determine that they are not the parents of node C, since $\angle RCA < \angle RCB$ and $\angle RCA < \angle RCD$.

Delaunay triangulation, each node can locally determine its child nodes with respect to a given tree, using its own coordinates, the coordinates of its neighbors, and the coordinates of the sender.

Local forwarding decisions at nodes are done using *compass routing* [15]. The basic building block of compass routing is that a node A, for a root node R, computes a node B as the parent in the tree, if B is the neighbor with the smallest angle to R. This is illustrated in Figure 2. Compass routing in general planar graphs may result in routing loops [14]. However, compass routing in Delaunay triangulations does not result in loops [15]. 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 a neighbor C is a child node with respect to a tree with root R, if the edge $\overline{AC}$ is a border of two triangles, say $\triangle ABC$ and $\triangle ACD$ (see Figure 3), and 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.

2.3 Building Delaunay Triangulations with Local Properties

Delaunay triangulations can be defined in terms of a locally enforceable property. A triangulation is said to be *locally equiangular* [23] if, for every convex quadrilateral formed by triangles $\triangle acb$ and $\triangle abd$ that share a common edge $\overline{a\,b}$, the minimum internal angle of triangles $\triangle acb$ and $\triangle abd$ is at least as large as the minimum internal angle of triangles $\triangle acd$ and $\triangle cbd$. This is illustrated in Figure 4. In [23] 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 formed by N and its neighbors. In Figure 5, node N can detect that the locally equiangular property is violated for triangles $\triangle NBC$ and $\triangle NCD$, and that the edge $\overline{N\,C}$ should be removed and replaced by an edge $\overline{D\,B}$. Thus, N can remove node C as one of its neighbors.

The protocol described in Section 4 builds and maintains a Delaunay triangulation overlay by enforcing the locally equiangular property for each node and its neighbors.

3 Comparative Evaluation of Overlay Topologies

We next present a comparative evaluation of the Delaunay Triangulation and other overlay network topologies for multicasting. The basis for our evaluation is a software tool for network topology generation [2].

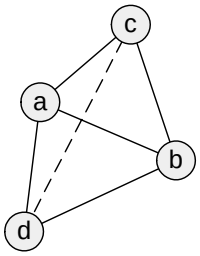


Figure 4: Locally equiangular property. The property holds for triangles $\triangle abc$ and $\triangle abd$ if the minimum internal angle is at least as large as the minimum internal angle of triangles $\triangle acd$ and $\triangle cdb$.

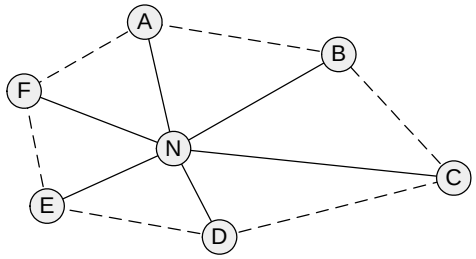


Figure 5: Locally equiangular property for a node N and its neighbors. Node N can enforce the equiangular property for all quadrilaterals formed by N and its neighbors A, B, C, D, E, and F. Here, N can determine that the locally equiangular property is violated for triangles $\triangle NBC$ and $\triangle NCD$. Thus, the edge $\overline{N\,C}$ should be replaced by an edge $\overline{D\,B}$.

We present results for a *Transit Stub* topology, which generates a network with a 2-layer hierarchy. The parameters for the topology are as follows. The network consists of four transit domains, each with 16 routers. The routers of each transit domain are randomly distributed over a 1024 by 1024 grid. There are 64 stub domains, each with 15 routers spread over a 32 by 32 grid. Each stub domain is connected to a transit domain router. Links between routers in transit and stub domains are set using the Waxman method [2]. The average number of links per router is approximately three. Hosts are connected to a router of a stub domain and are distributed over a four by four grid with the stub domain router at the center of the grid. The total number of hosts in the entire network is varied from two to 512 hosts, and incremented by powers of two. The hosts are distributed uniformly over the stub domains.

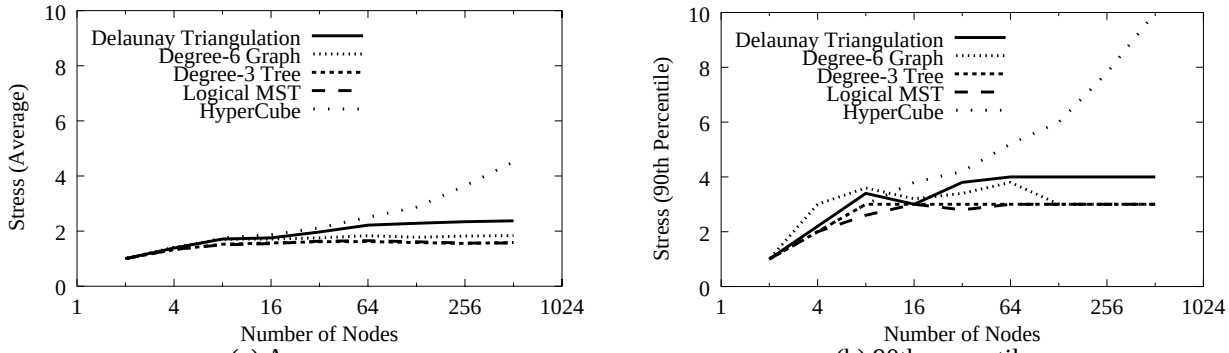
We assume that all unicast traffic is carried on the shortest-delay path between two hosts, where the delay between two hosts is determined by the length of the shortest path in the generated topology. For each generated network topology, we construct a set of overlay networks. Each host participates in an overlay network as a single node. We consider the following overlay topologies.

1. The *Delaunay triangulation* as described in Section 2. The coordinates of the nodes are the grid coordinates of the hosts in the generated graph.

2. A *Minimum Spanning Tree (MST)* is an overlay network which builds a shared tree with minimum total delay, as is done in the ALMI protocol [19].

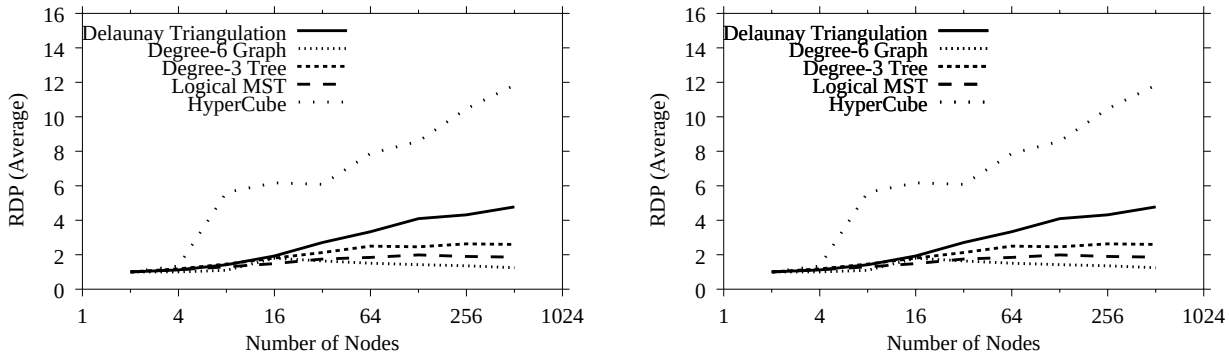
3. A *Degree-3 minimum spanning tree* represents a topology which is generated by the Yoid protocol [9]. In this overlay, each node has at most three links. We select a binary tree as an initial topology and use the update procedures described in [9] to improve the tree. We use overlay networks that are the result of 720 rounds of updates.

4. A *Degree-6 graph* is an overlay network that is created by the Narada protocol [4]. The algorithm establishes a mesh network where each node has at most six logical links. For multicast delivery, the method uses a DVMRP routing algorithm for building per-source trees [6]. The protocol performs



(a) Average. (b) 90th percentile.

Figure 6: Stress.



(a) Average. (b) 90th percentile.

Figure 7: Relative Delay Penalty (RDP).

periodic unicast delay measurements and improves the mesh, based on these measurements. We show results for overlay networks which are obtained after 720 rounds of improvements.

5. The *Hypercube* assigns nodes a binary string and arranges nodes in a logical incomplete hypercube. This topology completely ignores the network topology [16]. Data is disseminated using trees which are embedded in the hypercube [17].

For a performance comparison of overlay networks we use the performance metrics *relative delay penalty (RDP)* and *stress*, which have been used in the related literature (e.g., [4]).

- The *relative delay penalty (RDP)* for two hosts is the ratio of the delay in the overlay to the delay of the shortest-delay unicast path.
- The *stress* of a network-layer link is the number of identical copies of a packet that traverse the link for a given spanning tree embedded in an overlay network.

For network-layer multicasting, e.g., IP multicast, both RDP and link stress are equal to one. We point out that there are many other measures which can be used to evaluate overlay networks, such as the robustness of the topology to link or node failures, the speed of convergence of the overlay topology, and the overhead of the routing protocol in terms of computation and bandwidth needs.

It is important to note that the results for stress and relative delay penalty are dependent on the randomly generated network topology. To account for some of the randomness we present all numerical data as averages from five randomly generated network topologies, where identical parameters are used for each network topology.

In Figure 6 we show the stress values for various overlays when the number of hosts is varied between two and 512. The results show the average values (Figure 6(a)), and the 90th percentile values (Figure 6(b)) for the stress of links. With exception of the hypercube topology, all overlay topologies show similar values for stress. Figure 7 depicts the Relative Delay Penalty (RDP) values for all pairs of hosts, as averages (Figure 7(a)), and 90th percentile values (Figure 7(b)). The results show that overlay networks which take into consideration the network-layer topology incur a lower RDP than Delaunay triangulations. In summary, we observe that tree-based or mesh-based overlays improve the mapping of the logical overlay network to the network topology, if compared to Delaunay triangulations and hypercubes. On the other hand, considering that the presented Delaunay triangulations merely account for the geographical position of a node, but do not perform delay measurements between nodes in the overlay network, the results for Delaunay triangulations are encouraging.

4 The DT Protocol

In this section, we describe a network protocol which establishes and maintains a set of applications in a logical Delaunay triangulation. The protocol, referred to as DT (Delaunay Triangulation) protocol, has been implemented and tested as part of the HyperCast overlay software [10][2].

Essentially, the network protocol implements a distributed incremental algorithm for building a Delaunay triangulation.

In the following sections, 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. The length of x and y coordinates is set to 32 bits each. The logical address of a node is a configuration parameter, and can be assigned to a node, or derived from the geographical location or 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 $x_A < x_B$.

4.1 Neighbors and Neighbor Test

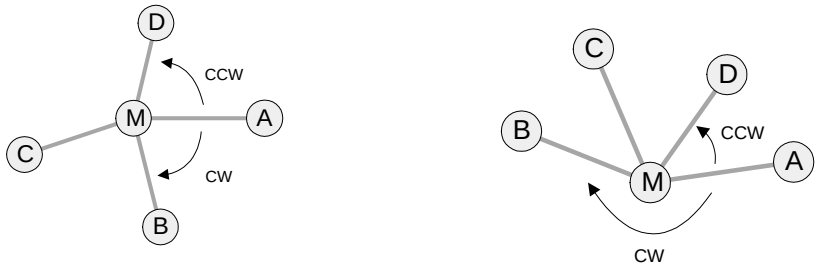
We say two nodes are neighbors if the edge connecting the two nodes appears in the Delaunay Triangulation graph. Each node maintains a *neighborhood table* which contains its neighbors in the Delaunay Triangulation overlay.

The protocol operations at a node mainly consists of adding and removing neighbors to and from its neighborhood table. To add or remove a node to or from its neighborhood table, a node needs to know if that node is eligible to be its neighbor in the overlay network topology. We next describe the *neighbor test* algorithm we developed for this purpose. The *neighbor test* is based on the *locally equiangular* property described in Subsection 2.3.

Before describing this algorithm, we first introduce the notions of clockwise (CW) and counter-clockwise (CCW) neighbors of a given node, say node A, with respect to another node, say node B. A neighbor of node A is said to be the CW (or CCW) neighbor with respect to node B, if (1) it forms the smallest clockwise or counter-clockwise angle to node B with node A as the pivot, and (2) the smallest clockwise or counter-clockwise angle is less than 180 degrees. The notions of CW and CCW neighbors are illustrated in Figure 8, where we show the CW and CCW neighbors of node M with respect to node A. In Figure 8(a),

[2] In addition to building overlay networks, the HyperCast software provides a socket-style API for transmitting data in an overlay network and tools that collect data from the nodes of an overlay network.

node B is the CW neighbor of M with respect to A, and is denoted as $B = CW_A(M)$. On the other hand, in Figure 8(b), node B is not regarded as the CW neighbor since the clockwise angle is larger than 180. In this case, we say node M has no CW neighbor with respect to node A. In both Figures 8(a) and (b), node D is the counter-clockwise neighbor of M with respect to A, and is denoted as $D = CCW_A(M)$.



(a) CW angle $< 180°$. (b) CW angle $\geq 180°$.

Figure 8: The CW and CCW neighbors of M with respect to A are M's neighbors that form the smallest CW and CCW angles to A, taking M as the pivot. In (a), $B = CW_A(M)$ and $D = CCW_A(M)$; In (b), node B is not the CW neighbor since the clockwise angle is greater an $180°$.

We now describe the *neighbor test*. 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.[3]

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, $CW_A(M)$ and $CCW_A(M)$, and the quadrilateral formed by M, $CCW_A(M)$, A, and $CW_A(M)$ is convex, A passes the neighbor test at M, if the edge $\overline{MA}$ 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 cannot 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 M 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 9(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 9(b). Note that this includes the case where M has neither a CW nor a CCW neighbor with respect to A.

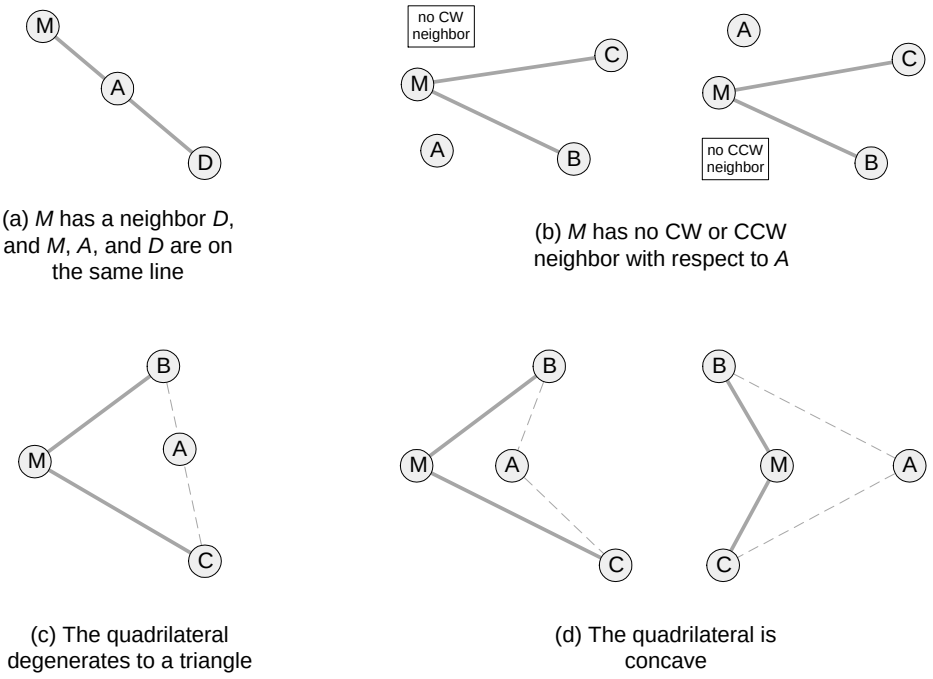
3. If the quadrilateral formed by M, $CCW_A(M)$, A, and $CW_A(M)$ is a triangle (see Figure 9(c)) or is concave (see Figure 9(d)), A passes the neighbor test at M.

As described above, A fails the neighbor test at M only in two cases: (1) M has a neighbor, say node D, and M, A, and D are on a line, and D is closer to M than A; (2) The *locally equiangular* property is violated in the convex quadrilateral formed by M, $CCW_A(M)$, A, and $CW_A(M)$. For all other cases, A passes the neighbor test at M.

We can argue the correctness of the neighbor test as follows. The neighbor test is a consequence of the *locally equiangular* property from [23], which states that a triangulation where all convex quadrilaterals

[3]For conciseness, we use "node" and "coordinates of a node" interchangeably.

are locally equiangular, is a Delaunay Triangulation. The neighbor test enforces the property from [23] by enforcing two points: (1) Whenever a convex quadrilateral is formed by the nodes M, $CCW_A(M)$, A, and $CW_A(M)$, then the locally equiangular property is enforced; (2) When no convex quadrilateral can be formed by nodes M, $CCW_A(M)$, A, and $CW_A(M)$, 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.



(a) *M* has a neighbor *D*, and *M*, *A*, and *D* are on the same line

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

(c) The quadrilateral degenerates to a triangle

(d) The quadrilateral is concave

Figure 9: *Neighbor test* at the testing node M for a tested node A, for cases where the *locally equiangular* property is not applicable, since it is not feasible to form a convex quadrilateral of nodes M, A, $CW_A(M)$, $CCW_A(M)$. The depicted scenarios show cases when A is added as a neighbor of M. Solid lines show M's current neighbors, and dashed lines are used to indicate the quadrilateral.

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.[4]

Each entry of the neighborhood table at a node has columns for the neighbor, the CW neighbor, and the CCW neighbor. For example, the neighborhood entry at node M for one of its neighbors A has the form

Neighborhood table at node M:

Neighbor	CW neighbor	CCW neighbor
A	$E = CW_M(A)$	$F = CCW_M(A)$
...	...	...

Now, when a node M receives a neighbor message from node A, it first checks if node A is in its neighborhood table. If node A is, then it updates the entry of node A in the neighborhood table, changing the CW

[4] The receiver of a neighbor must know the physical address of the sending node. However, if the physical address of the sender is included in the payload of a neighborhood message, the DT protocol cannot be used across a private/public network boundary. Hence, it is preferred that the physical address of the sender of a neighbor message is recovered by the receiver of the message, using the underlying transport protocol.

and CCW neighbor fields of the entry if they are different from the CW and CCW neighbor fields contained in the neighbor message. If node A is not a neighbor of M, then M runs a *neighbor test* for A. If A passes the neighbor test, it will be added as a new neighbor in the neighborhood table of M.

We say a node is a *candidate neighbor* of another node, if it is not in the neighborhood table of that node and it passes the neighbor test at that node. The formal definition of the *candidate neighbor* will be given in Subsection 4.6. A node can learn about a candidate neighbor only in two ways: (1) through NewNode messages described in Subsection 4.4 and (2) from the CW or CCW neighbor fields in the neighborhood table. If a node has candidate neighbors, it will send a neighbor message to the closest candidate neighbor. If the candidate neighbor responds with a neighbor message, the candidate neighbor is elevated to the status of a neighbor, i.e., a new entry is created in the neighborhood table.

A neighbor can be removed from the neighborhood table for any one of the following reasons: (1) the neighbor has sent a message indicating 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.

We say that a node A is *stable*, if all nodes that appear in the CW or CCW neighbor columns of A's neighborhood table also appear in the neighbor column of A's neighborhood table. This means that node A has knowledge of all nodes in the overlay network that will pass its *neighbor test*. If all nodes are stable, then the set of nodes has established a logical Delaunay triangulation.

4.2 Rendezvous Mechanisms with DT Servers and Leaders

Any protocol that builds an overlay network must provide a rendezvous mechanism that enable nodes which are not members of the overlay to communicate with nodes in the overlay. The rendezvous mechanism is also used when new nodes join an overlay and when the overlay network has been partitioned and must be repaired.[5]

One can think of three methods that can accomplish a rendezvous between members and non-members of an overlay network: (1) Non-members have available a broadcast mechanism to announce themselves to members of the overlay network, (2) non-members maintain a list of 'likely' members of the overlay network (a 'buddy list') and contact members from this list, (3) non-members contact a well-known server to learn about members of the overlay network.

In the DT protocol, we select the third method, i.e., members join the overlay and partitioned overlay networks are repaired with the help of a server. A reservation against using a well-known server is that the server may become a performance bottleneck, and that it constitutes a single point of failure. To address the performance concern, 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. The single point of failure can be avoided by adapting the DT protocol so that it supports multiple servers. Also, we emphasize that the effort to create variations of the DT protocol which use broadcast announcements or buddy lists is rather small, and that these variations can preserve the main characteristics of the DT protocol.

The server component of the DT protocol is called *DT server*. New nodes join the overlay network by sending a request 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, and, thus, establishes communication with some node in the overlay network.

The DT server is also engaged in repairing partitions of the overlay network as follows. In the DT protocol, a node believes to be 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 believes to be a Leader. If the server receives messages

[5]We say an overlay network is partitioned if the graph represented by the overlay has multiple connected components. In a partitioned overlay, some nodes cannot communicate with each other across the overlay.

from multiple Leaders, the server replies with the identity of the Leader with the greatest coordinates. By virtue of the Delaunay triangulation, if a node A that believes to be a Leader, learns about a node B and $coord(A) < coord(B)$, then B will pass A's neighbor test, and, consequently, A adds node B as a candidate neighbor. Also, A no longer believes to be a Leader. Thus, the partition of the overlay network is repaired.

The DT server maintains a list ('cache') of 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 newly joining node, this address is taken from the cache. If the cache is empty, then the DT server returns the address of the Leader. We set the default size of the cache to 100 nodes. If a newly joining node contacts the DT server and the cache is not full, this node will be added to the cache. The DT server periodically queries nodes in the cache to verify that these nodes are still members of the overlay. If a node does not respond to a query it will eventually be removed from the cache. Also, a node is removed from the cache after the DT server has selected this node six times as the 'contact node' for a newly joining node.

4.3 Timers

The DT protocol is a soft-state protocol, that is, all remote state information is periodically refreshed, and is invalidated if it is not refreshed. The operations to recalculate and refresh state are triggered with the help of timers. A node of the DT protocol uses the following three timers.

Heartbeat Timer. The heartbeat timer determines when a node sends neighbor 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. Otherwise, it is in *SlowHeartbeat* mode. The operation of the heartbeat timer in two modes trades off the need for fast convergence of the overlay network when the topology changes, and low bandwidth consumption in a steady state. In our experiments, we set the timeout value of the heartbeat timer to $t_{SlowHeartbeat} = 2$ seconds in *SlowHeartbeat* mode and to $t_{FastHeartbeat} = 0.25$ seconds in *FastHeartbeat* mode.

Neighbor Timer. If a node has not received a neighbor message from one of its neighbors for $t_{Neighbor}$ seconds, the neighbor will be deleted from the neighborhood table. There is one Neighbor Timer for each neighbor in the neighborhood table. The default timeout value of the neighbor timer is set 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 timeout value of Backoff timer is set to $t_{Backoff} = t_{FastHeartbeat}$ and doubled after each repeated transmission, until it reaches $t_{Backoff} = t_{Neighbor}$ (= 10 seconds). If there are alternate DT servers, the node switches to an alternate DT server when $t_{Backoff} \geq t_{Neighbor}$ seconds.

The DT server keeps 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 $t_{Leader} = 10$ seconds.

4.4 Message Types

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

the transmission and reception of each message. The message format is discussed in Appendix B.

HelloNeighbor and HelloNotNeighbor Messages. These messages are used to create and refresh neighborhood tables at nodes. Each HelloNeighbor[6] and HelloNotNeighbor message contains the logical and physical addresses of the sender[7], 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 a 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, the message is sent to the candidate neighbor with the 'closest' coordinates.

A HelloNotNeighbor message is sent as an immediate reply to the reception of a HelloNeighbor message from a node that fails 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.

A special case exists, when several nodes have the same logical address. When a node A learns about the existence of a node B with $coord(A) = coord(B)$, through any method except a HelloNeighbor message from B, then A sends a HelloNeighbor message to B. When a node B receives a HelloNeighbor message from a node A, and $coord(B) = coord(A)$, then B changes its logical address. When a node A has a node B in its neighborhood table, and A learns about a node C, with $coord(B) = coord(C)$, then A sends a HelloNotNeighbor message to C. This is discussed in more detail in Subsection 4.5.

Goodbye Message. When a node leaves the overlay, it sends Goodbye messages to the DT server and all its neighbors. If a node receives a Goodbye message, it removes the sender of the Goodbye message from the 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 each message received, until the process that runs the node is terminated by the application.

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

Each ServerRequest message contains the logical and physical addresses of the sender[8]. 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 $coord(X) \leq coord(Y)$. Newly joining nodes use addresses in the ServerReply message to find a node that is already in the overlay. Leaders use the addresses in the ServerReply 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 from the DT server the address of some node in the overlay, say D, 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. In such a way, the NewNode message is routed through the overlay towards the coordinates of the new node, until the NewNode message reaches a node where the new node passes a neighbor test.

[6]Earlier, we referred to HelloNeighbor messages as *neighbor messages*.

[7] Refer to Footnote 4. The receiver of a HelloNeighbor or HelloNotNeighbor message needs to know the physical address of the sending node. However, it is preferred that the physical address of the sender is recovered by the receiver of the message, rather than being included in the message itself.

[8] See Footnote 7. The same argument holds for the sender of a ServerRequest message. The physical address of the sender should be recovered by the receiver of the message, rather than being included in the message itself.

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 every node in the cache. A node that receives a CachePing message immediately replies with a CachePong message.

4.5 Shifting Coordinates

Since the logical address of a node is a configuration parameter, it may happen that two nodes have the same coordinates, or that the coordinates of four nodes lie on a circle. In the former case, the Delaunay triangulation is not defined, and in the latter case, the Delaunay triangulation overlay is not unique. In both cases, the DT protocol forces one of the nodes to change its coordinates by a small amount, thus, ensuring 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, and 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 node B, then node A sends 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 $CCW_N(A)$, contained in the message lie on a circle, A will shift its coordinates before processing the message.

Each time a node receives a HelloNeighbor or HelloNotNeighbor message from a neighbor, it checks if the neighbor's logical address has changed. If the logical address has changed, the node removes the neighbor's entry from the neighborhood table and then processes the message. In most cases, the node with the shifted coordinates will be added again as a neighbor.

4.6 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 for the DT server. A detailed description of the state transitions is presented in tabular form in Appendix A.

4.6.1 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 node.

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 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 *Leader 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 will generate NewNode Messages when it learns about a node in the overlay to contact. A node with neighbors does not send NewNode Messages.

The definitions of the five states are given in Table 1.

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 also appear in the neighbor column; Otherwise node X is not stable. We say a node M has a candidate neighbor, say node N,

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 1: Node State Definitions.

if (1) N appears in the CW or CCW column of M's neighborhood table, or if N 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 2.

Sub-state Name	State Definition
Stable Without Candidate Neighbor	The node is stable and has no candidate neighbors
Stable With Candidate Neighbor	The node is stable and has candidate neighbors
Not Stable	The node is not stable

Table 2: Node Sub-state Definitions.

A new node starts in state **Stopped**. When it is in state **Leader With Neighbor** and **Not Leader**, the node also has a sub-state. The transition diagram of states and sub-states is shown in Figure 10.

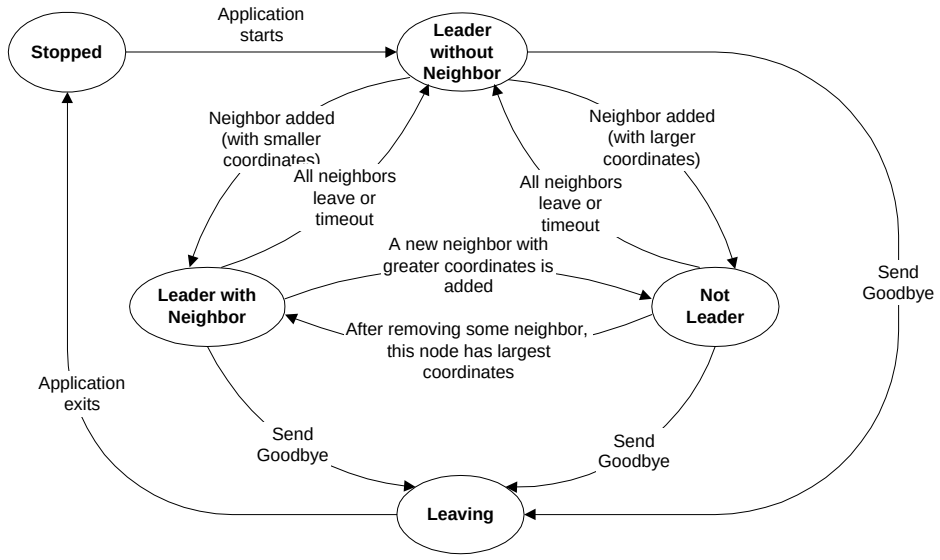
4.6.2 DT Server States

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.

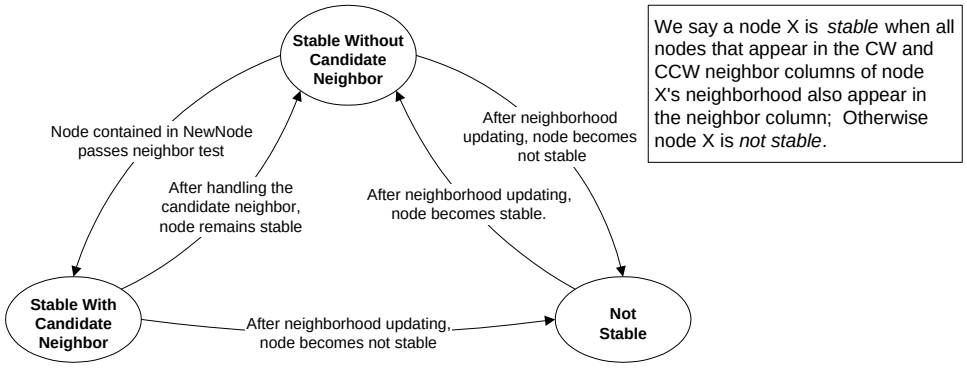
State Name	State Definition
Has Leader	The node cache contains at least one node
Without Leader	The node cache is empty

Table 3: DT Server State Definitions.

The state transition diagram of the DT server is shown in Figure 11. The DT Server starts in state **Without Leader**. 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 Leader**.



(a) Transition Diagram of Node States.



(b) Transition Diagram for Sub-states.

Figure 10: State transition diagrams for nodes. States are indicated as circles. State transitions are indicated as arcs. Each arc is labeled with the condition that triggers the transition.

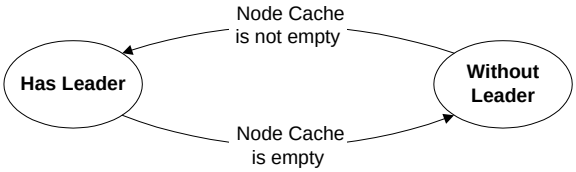


Figure 11: DT Server State Transition Diagram.

4.7 Examples

In an overlay network, without any changes 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; and (3) the server exchanges CachePing and CachePong messages with the nodes in its cache every $t_{SlowHeartbeat}$ seconds.

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

4.7.1 Node Joins

In Figure 12 we 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 12(a), N first sends a ServerRequest to the DT server, and receives a ServerReply, which contains the logical and physical addresses of some node X with $coord(X) > coord(N)$. Then, node N sends a NewNode message to X (Figure 12(b)). X performs a neighbor test for N, which fails. Therefore, X forwards the NewNode message to neighbor Y, which is closer to N than X. Assuming that N fails the neighbor test at Y, node Y forwards the NewNode message to D which is closer to N than Y. At node D, N passes the neighbor test and, therefore, D makes N a candidate neighbor and sends a HelloNeighbor message to N.[9] Now, N has found its first neighbor.

Since the HelloNeighbor from D in Figure 12(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 message to its neighbor D, and its closest candidate neighbor B (Figure 12(c)). As soon as these HelloNeighbor messages are received at B and D, these nodes will drop each other from their neighborhood table. In other words, the link in the overlay between nodes B and D is removed.

In Figure 12(d) we assume that the Heartbeat timers expire 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.) Both nodes send HelloNeighbor messages to 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 CW or CCW neighbor. Hence, N becomes a candidate neighbor at 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 their candidate neighbor N (Figure 12(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 12(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 joining node N to the overlay network. Subsequently, each node sends HelloNeighbor messages to its neighbors at each Heartbeat

[9] Since the NewNode message contains the physical address of node N, any node that receives the NewNode message can send messages to N.

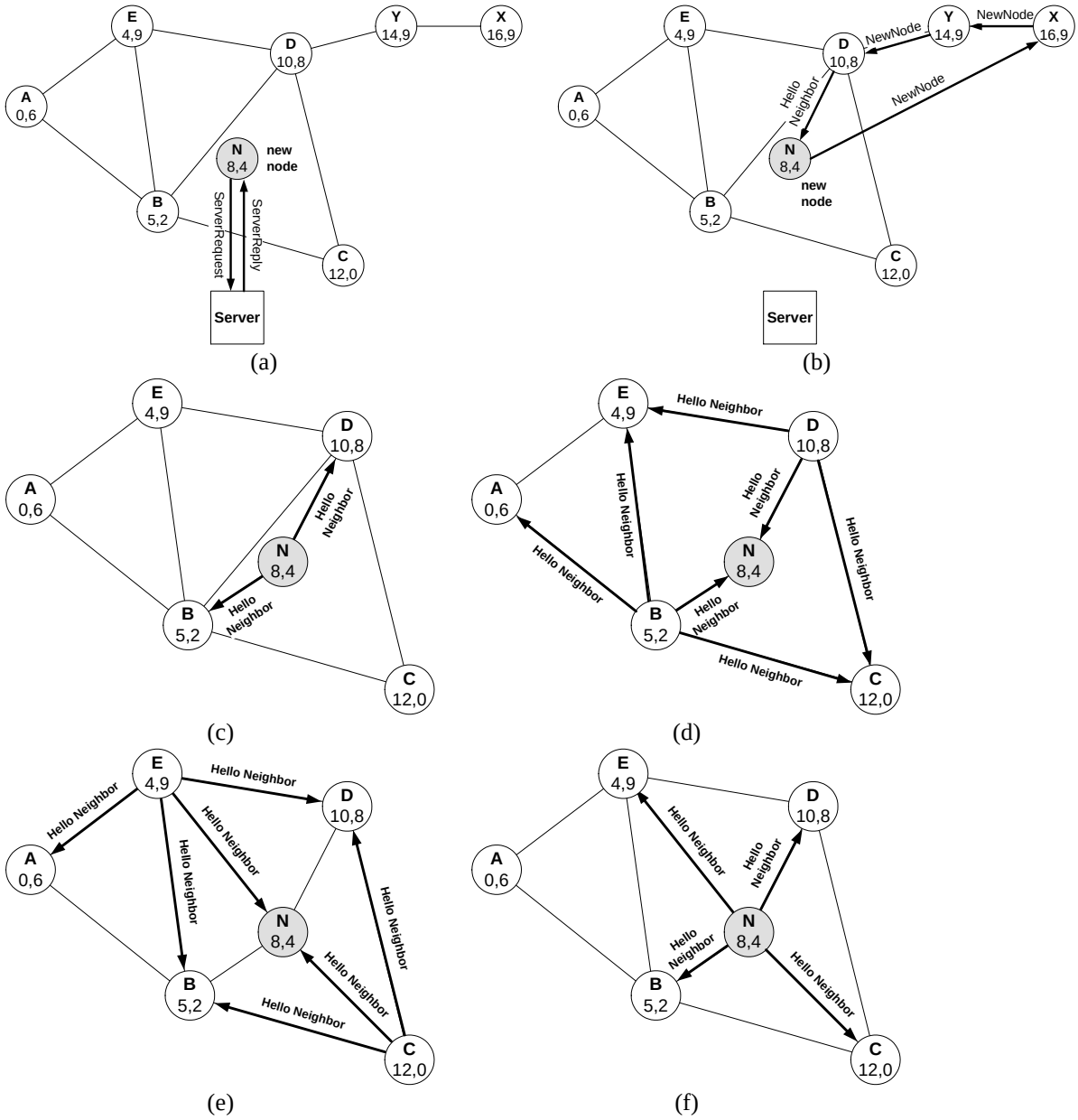


Figure 12: Node N with coordinates $coord(N) = (8, 4)$ joins the overlay network. Note that in (a) and (b), we have omitted some edges from the Delaunay triangulation for the sake of simplicity, and in (c)–(f), we have omitted nodes X and Y.

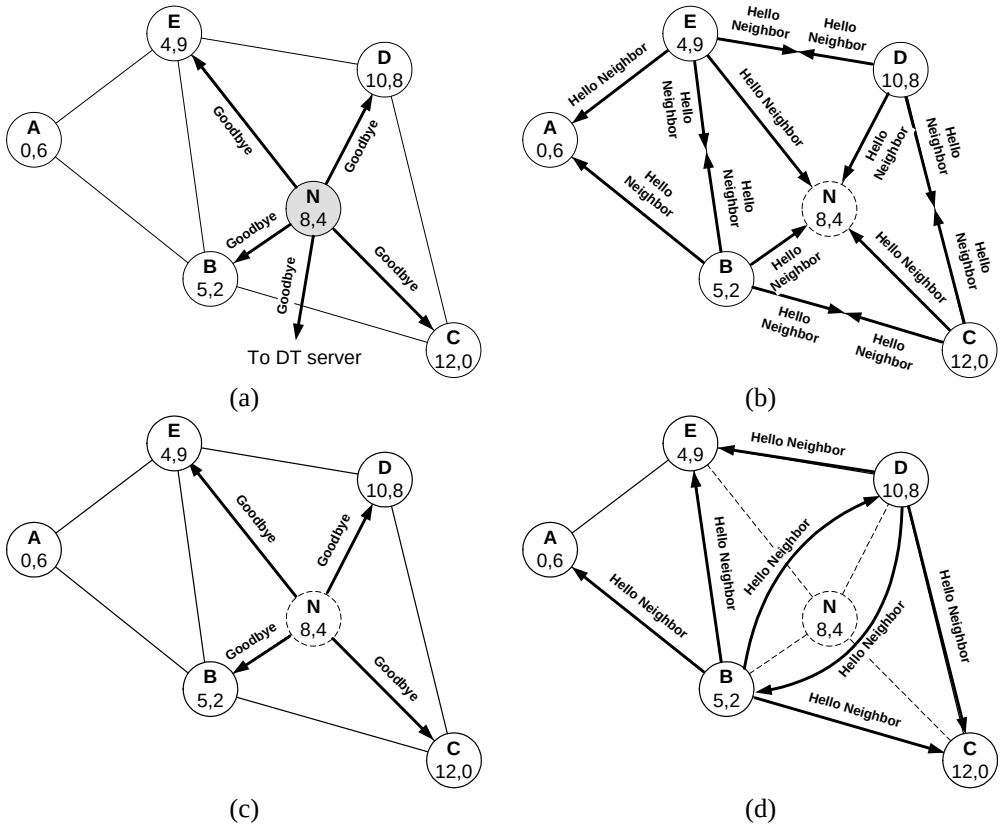


Figure 13: Node N with coordinates $coord(N) = (8, 4)$ leaves the overlay network.

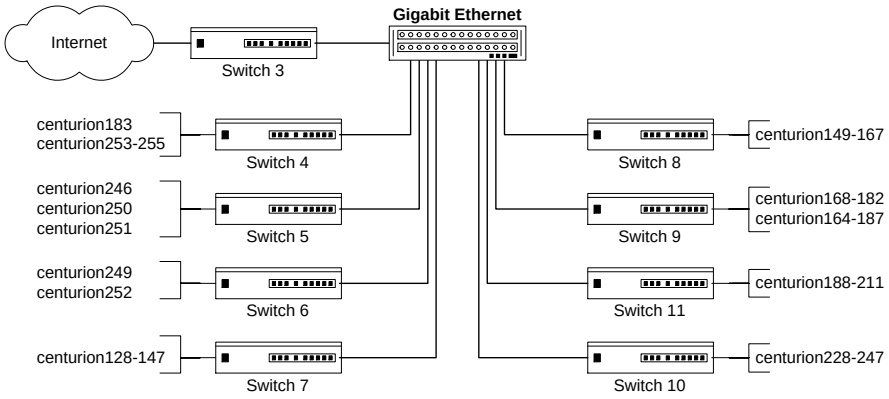


Figure 14: Network topology of the Centurion cluster. The figure only shows equipment involved in the experiments. Switches 3 to 11 are Ethernet Switches with 100 Mbps ports and one 1 Gbps uplink. All links are full-duplex. Hosts are labeled *'centurionN'*, where N is a number. All hosts have valid IP addresses and may run other applications at the time of the experiments. At most 100 nodes are involved in any single experiment.

timeout.

4.7.2 Node Leaves

In Figure 13 we illustrate the steps involved when node N leaves the overlay network. When N decides to leave the overlay, it sends Goodbye messages to all its neighbors and the DT server (Figure 13(a)). When the server receives the Goodbye message, it removes N from the cache. When the neighbors receive the Goodbye message, they remove N from the neighborhood table.

However, even though N is deleted as neighbor at nodes B, C, D, and E, these nodes have some other neighbor entries, where N is listed as 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. By definition of a candidate neighbor, N is now a candidate neighbor at B, C, D, and E, and the nodes will send HelloNeighbor messages to N at their next Heartbeat timeout.[10]

Let us now assume that all nodes send HelloNeighbor message to their neighbors, and their candidate neighbor N (Figure 13(b)). When N receives the messages, it responds with Goodbye messages, as shown in Figure 13(c).

The HelloNeighbor messages sent in Figure 13(b) contain the updated values of the CW and the 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, that is, $C = CW_E(B)$. As a result, after Figure 13(b), node N no longer exists as a CW or CCW neighbor in the neighborhood tables of any node. Further, nodes B, C, D, and E, know about each other either as neighbors, or as CW or CCW neighbor of some neighborhood table entry. When the neighbor tests are executed, C fails 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 13(d)). Once these messages are received, both B and D have established each other as neighbors, and, as a result, the overlay network is repaired.

[10]These HelloNeighbor messages to N are superfluous. However, avoiding these messages adds a requirement that nodes remember recently received Goodbye messages.

5 Evaluation of the DT Protocol

We have evaluated the performance characteristics of the DT protocol in measurement experiments on a cluster of Linux PCs. The experiments include up to 100 PCs and overlay networks with up to 10 000 nodes.

The DT protocol was implemented in Java using Sun's Java Virtual Machine 1.3.0, which includes the HotSpot just-in-time compiler. Details of the implementation can be obtained from [10].

The measurement experiments were conducted on the Centurion computer cluster at the University of Virginia. The experiments use up to 100 PCs from the cluster, each equipped with two 400 Mhz Pentium II processors and 256MB RAM. The server of the DT protocol was run on a 933 Mhz Pentium III with 1GB RAM. All machines run the Linux 2.2.14 operating system. The network of the Centurion cluster is a switched Ethernet network with a two-level hierarchy, as shown in Figure 14. Each PC is connected to a 100 Mbps port of an Ethernet switch. Each Ethernet switch has a 1 Gbps uplink to a Gigabit Ethernet switch. All machines of the Centurion cluster have valid IP addresses and are on the same IP subnetwork. Hosts used in the experiments were not available for exclusive use, and may have run other applications.

In the experiments, each PC (henceforth referred to as "host") executes between 1 and 100 nodes of the overlay networks, allowing overlay networks with up to 10 000 nodes. The number of nodes in an experiment are evenly distributed over the 100 hosts of the PC cluster. For instance, in an experiment with 1 000 nodes, 10 nodes are assigned to each host. In all experiments, the coordinates of a node in the DT triangulation are assigned as a randomly selected point from a 10 000 by 10 000 grid. Results obtained with this random assignment of coordinates can be interpreted as lower bounds for the expected performance of any more sophisticated assignment method that considers the topology of the underlying network.

5.1 Evaluation of the Overlay Graphs

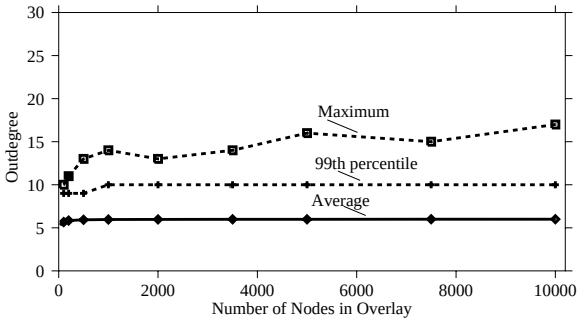
We first evaluate the properties of the overlay graphs generated by the DT protocol for the network topology in Figure 14. We have constructed overlay networks with 1 000 to 10 000 nodes. Recall that the coordinates of nodes are randomly assigned, and, hence, the DT protocol does not consider the network topology when constructing an overlay network.

We evaluate the *outdegree*, defined as the number of neighbors of a node, the *path length*, defined as the number of logical overlay edges between a given pair of nodes, and the *stress*, where the stress of a network link l with respect to a multicast sender is the number of overlay edges in the embedded spanning tree (with the multicast sender as root of the tree) that pass over link l. Since the outdegree of a node provides an upper bound for the number of times that any particular message is forwarded at a node, the outdegree is a measure for the processing load at a node in the overlay. The path length is an indicator for the delay in the overlay network. The stress indicates how efficient the overlay utilizes the available network bandwidth.

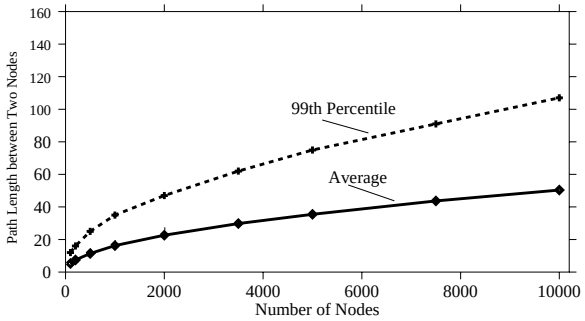
The results are shown in Figure 15. Each data point in the plots contains the results from five generated overlay networks. Figure 15(a) shows the outdegree of nodes when the number of nodes in the overlay network is increased. The average outdegree is approximately six, as is expected for any triangulation graph. Although the worst-case outdegree of a node in a Delaunay triangulation is $N - 1$, where N is the number of nodes, the maximum outdegree is relatively small.

Figure 15(b) shows the path length in the overlay network between randomly selected pairs of nodes as a function of the number of nodes in the overlay topology. Each data point in the graph shows the values from five topologies, where in each topology, we evaluate 1 000 randomly selected pairs of nodes. (The random selection of paths is justified by the considerable effort to compute all paths.) The average number of hops closely matches $\sqrt{N/4}$, where N is the number of nodes.

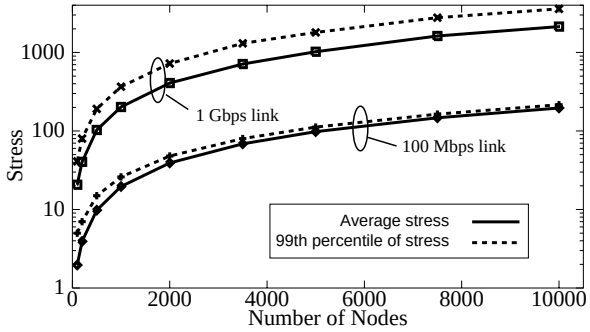
Figure 15(c) shows the stress values for the links of the network in Figure 14. (We ignore any logical



(a) Outdegree of nodes.



(b) Path Length between Two Nodes.



(c) Stress.

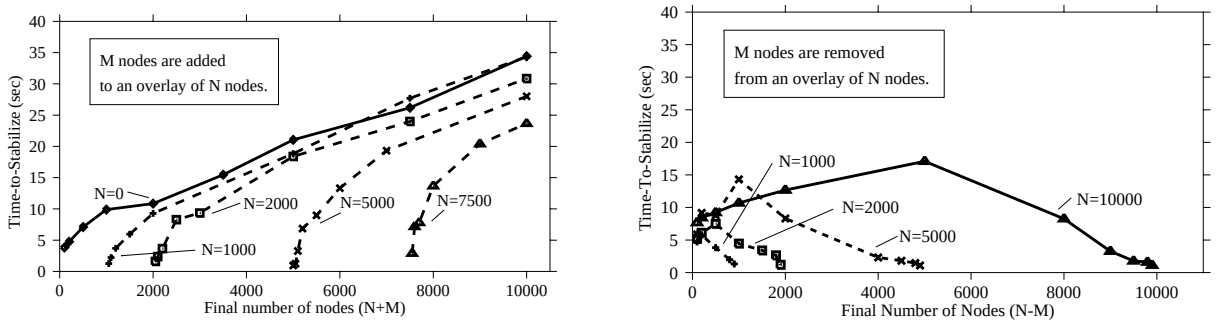
Figure 15: Outdegree of nodes, path length between two nodes, and stress of links in the overlay network. Each data point shows the results from five generated graphs. In (b), the values for the path length are computed from 1 000 randomly selected node pairs. In (c), the values for the 'stress' of a link are computed for at most 1000 randomly selected multicast trees.

links between nodes that run on the same host.) With the random assignment of logical addresses to nodes of the overlay, the links at the Gigabit switch are expected to have higher stress values than the 100 Mbps links. Thus, we plot the stress in Figure 15(c) separately for 100 Mbps links and 1 Gbps links. Each data point contains the results from five overlay topologies generated for the network in Figure 14. Since the values for stress at a link depend on the selection of the multicast routing tree, we calculate the stress for the multicast routing trees of 1 000 randomly selected senders (if the number of nodes is less than 1 000, we consider all senders). The results in Figure 15(c) show that the stress for 10 000 nodes can exceed 100 at a 100 Mbps link, and can exceed 1 000 at a 1 Gbps link. In comparison to Section 3, the stress values are significantly worse. This is due to the random assignment of (x,y) coordinates, and the star-shaped topology of the network in Figure 14. Note that the small difference between the average stress and the 99th percentile of the stress indicate that the stress is high for all multicast trees.

5.2 Performance of the DT Protocol

We now evaluate the performance of the DT protocol by measuring the time required to form a Delaunay triangulation overlay network, and to repair the Delaunay triangulation after nodes depart. Later, we examine the steady state bandwidth requirements of the protocol at the nodes and at the DT server.

To measure the time to build an overlay network, we set up an initial Delaunay triangulation overlay network with N nodes, and add M new nodes at the beginning of the experiment. The rate at which new nodes are generated is limited by the computational power and number of the hosts. In our experiments, each hosts spawns off new nodes as fast as possible. Then we measure the time delay until the overlay



(a) Time-to-Stabilize when new nodes join.

(b) Time-to-Stabilize when nodes leave.

Figure 16: Time-to-Stabilize when nodes join or leave the overlay network. In (a), M new nodes are added to an overlay of N nodes. The time to stabilize is the time until the resulting overlay network of $N + M$ nodes has formed a Delaunay triangulation. In (b), there are N nodes initially, and M nodes depart at the beginning of the experiment. Here, the time to stabilize is the time until the resulting overlay network of $N - M$ nodes has formed a Delaunay triangulation.

network of $N + M$ nodes has formed a Delaunay triangulation. The *Time-to-Stabilize* is measured from the time the first new node joins the overlay until the last node has been added. In all experiments, nodes are evenly distributed over 100 hosts of the PC cluster.

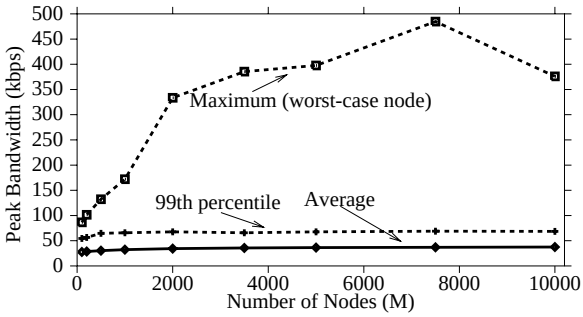
The measurements of delays and bandwidth requirements heavily depend on the selection of the timer values of the protocols. In all our experiments, we use the values given in Subsection 4.3.

Figure 16(a) shows the Time-to-Stabilize for experiments with $N = 0, 1\,000, 2\,000, 5\,000, 7\,500$ and $M = 50 - 10\,000$ nodes. We consider overlay networks with up to 10,000 nodes, that is, $M + N \leq 10\,000$. Each data point shows the average of five repetitions of the same experiment. Since the coordinates are randomly assigned, each repetition of an experiment is likely to result in a different overlay topology. Figure 16(a) contains five curves, one plot for each value of N. Each curve shows the Time-to-Stabilize, measured in seconds, as a function of $M + N$, i.e., the final number of nodes. The figure shows that a Delaunay triangulation with $10\,000$ nodes is formed in less than 35 seconds. In other words, on the average one node is added every 3.5 msec.

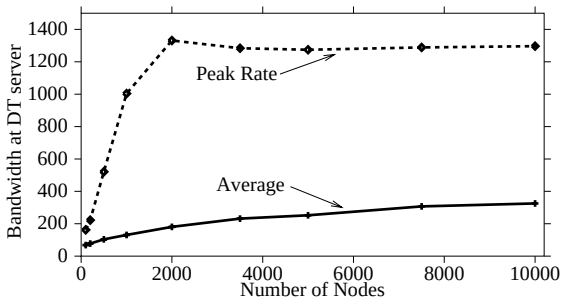
Figure 16(b) shows the time to stabilize when nodes leave the overlay network. We assume that there is an overlay network with N nodes, where $N = 1\,000, 2\,000, 5\,000, 10\,000$. At the beginning of each experiment, M nodes depart from the overlay network. We measure the time until the resulting overlay network of $N - M$ stabilizes to a Delaunay triangulation. Each data point presents an average of five repetitions of an experiment. As before, we present one curve for each value of N, and depict the Time-to-Stabilize as a function of $N - M$. The results show that the Delaunay triangulation is quickly repaired even when a large number of nodes depart. When almost all nodes leave the overlay network, that is, when $N - M$ is small, the overlay network stabilizes on the average in less than 10 seconds for all considered values of N.

We also measured the peak bandwidth requirements when nodes are added to an overlay network, by counting the maximum number of sent and received messages between any two expirations of the Heartbeat timer at nodes and the DT server. Recall that Hearbeat timers are set to either $t_{FastHeartbeat}$ or $t_{SlowHeartbeat}$ seconds. We present the data in terms of bits per second, but we only account for the payload size of messages and do not account for the UDP header, IP header, or Ethernet headers. Since messages are at most 61 bytes long, the bandwidth requirements at the data link layer are up to 75% higher than shown in the graphs.

Figure 17 shows the bandwidth requirements of nodes in an experiment which builds a Delaunay tri-

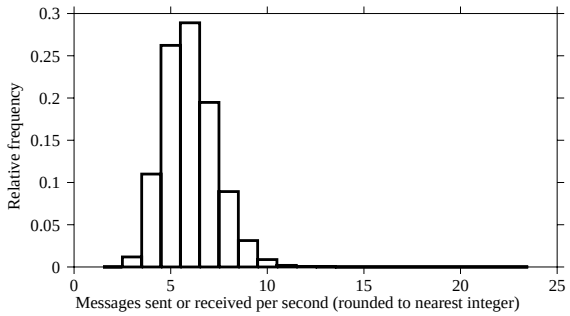


(a) Peak bandwidth requirements of nodes.

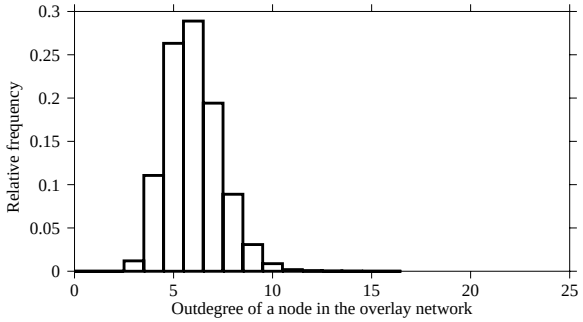


(b) Average and peak bandwidth requirements at DT server.

Figure 17: Bandwidth requirements of nodes and the DT server in an experiment where M nodes are added to an initially empty overlay network. Measurements are taken for intervals of the length of the Heartbat timer until the Delaunay triangulation is completed. In (a), the peak bandwidth of nodes is presented as average, 99th percentile, and maximum over all nodes. In (b), the average and peak bandwidth requirements at the DT server are presented.



(a) Messages Transmitted and Received.



(b) Number of Neighbors.

Figure 18: Distribution of the number of messages (sent and received) of a node and the number of neighbors per node in an overlay network with 10 000 nodes. The histogram in (a) shows the average number of messages sent and received by a node in a 1-second time interval.

angulation with M nodes, starting with an overlay network with $N = 0$ nodes. The figure depicts the bandwidth requirements as a function of M. The measurements are taken over the entire length of the experiment, that is, until the Delaunay triangulation of M nodes is completed. Figure 17(a) illustrates that the peak bandwidth, averaged over all nodes, is below 38 kbps for all values of M. The 99th percentile of the peak bandwidth at nodes is below 70 kbps. However, the node with the highest peak bandwidth, was measured at 485 kbps, corresponding to 249 messages in a time period of $t_{FastHeartbeat} = 0.25$ seconds.

Figure 17(b) shows the peak and average bandwidth requirements at the DT server during the same experiment. The average bandwidth requirements at the DT server are below 400 kbps, even when 10 000 nodes want to join the overlay network. The peak bandwidth requirement at the DT server is about 1 300 kbps. Since each node that joins the overlay network sends a message to the DT server, the bandwidth requirements at the DT server should increase with M. The fact that the peak bandwidth does not increase for $M \geq 2000$ nodes indicates that the rate at which new nodes are generated is limited by the rate at which the hosts can start new overlay nodes.

5.3 Bandwidth Requirements of DT Protocol in Steady State

We have performed measurements of the bandwidth requirements by the DT protocol in a steady state, that is, when no nodes are added or removed from the Delaunay triangulation overlay. We use an overlay

network with 10 000 nodes and measure the number of messages transmitted by each node. For each overlay network topology, we take measurements over a period of one hour. We repeat each experiment five times. Figure 18(a) shows a histogram which depicts the distribution of the number of transmitted and received messages for all five repetitions of the experiment. Since, in a steady state, the number of messages at nodes consists mostly of HelloNeighbor messages, the amount of traffic at a node largely depends on the outdegree of the node. To support this observation, we include in Figure 18(b) a histogram which depicts the distribution of the outdegree of nodes. A comparison of Figures 18(a) and 18(b) makes clear that the traffic at a node indeed correlates with the number of neighbors. Recall that in a triangulation graph, each node has on the average six neighbors. Thus, with the Heartbeat timer set to $t_{SlowHeartbeat} = 2$ seconds, we expect that each node sends and receives 12 messages over a period of 2 seconds. With 61 bytes per HelloNeighbor message (not counting UDP, IP and Ethernet headers), the bandwidth requirements per node are less than 3 kbps. In all measurements of overlay networks in steady state, no node sent or received more than 23 messages per second, or more than 11.2 kbps of traffic.

As a final comment, the steady state traffic at the DT server consists mostly of CachePing and CachePong messages to the nodes in the cache. If the cache has a size of 100 nodes, the steady state bandwidth requirement at the DT server is approximately 50 kbps.

5.4 Measurements of Multicast Bulk Transfers

We tested the application-level performance by measuring the throughput and delay of multicasting bulk data in the overlay network. Recall from Section 2 that a node A in a Delaunay triangulation overlay forwards a multicast message from source node S to a neighbor node B, if A is the next hop on the (compass routed) path from B to S. Thus, once the overlay network is established, no routing protocol is needed.

We present measurements from an experiment where multicast data is carried over (unicast) TCP links between neighbors in the overlay. We assume that all data is partitioned into messages with 16 kB payload and 16 bytes of header information.[11] The header information contains, among others, the logical address of the sender of the multicast message.

The measurements of multicast transmissions are performed for overlay networks with $N = 2$ to 1 000 nodes. In the measurement experiments we run 1 or 10 nodes on each host, and the number of hosts involved in an experiment is varied between two and 100. In experiments with one overlay node per host, the multicast sender performs a bulk transfer of 100 MB (= 6400 messages). For experiments with 10 nodes per host, the multicast sender performs a bulk transfer of 10 MB of data (= 640 messages).

We measure the throughput in multicast trees for 10 randomly selected senders, with the exception of $N = 2$, where we only have two senders. The throughput is measured at all receivers, as the ratio of the amount of data sent (10 MB or 100 MB) and the time lag between the receipt of the first and last message. As in all previous experiments, we repeat each measurement five times. Figure 19 shows the average throughput, averaged over all nodes in all multicast trees and all repetitions. To give an indication of the distribution of the throughput values, we include the range of throughput measurements for all multicast trees as error bars. Note that there is a different multicast tree for each generated overlay.

Given a network topology (in our case, Figure 14), an overlay network, and a multicast sender, the 'stress' (see Figure 6) imposes a limit on the maximum achievable throughout. The achievable throughput can be bounded by $\min_{l \in L} bw_l / s_l$, where L is the set of all network links in the overlay network which carry traffic from the multicast sender, bw_l is the capacity of link l (here, either 100 Mbps or 1 Gbps), and s_l is the *stress* of link l. We calculate these bounds and compare them to the measured values. We include the average value of the bounds, averaged over all multicast trees that were evaluated.

[11] A description of the message format used by the protocol is beyond the scope of this paper and we refer to [10].

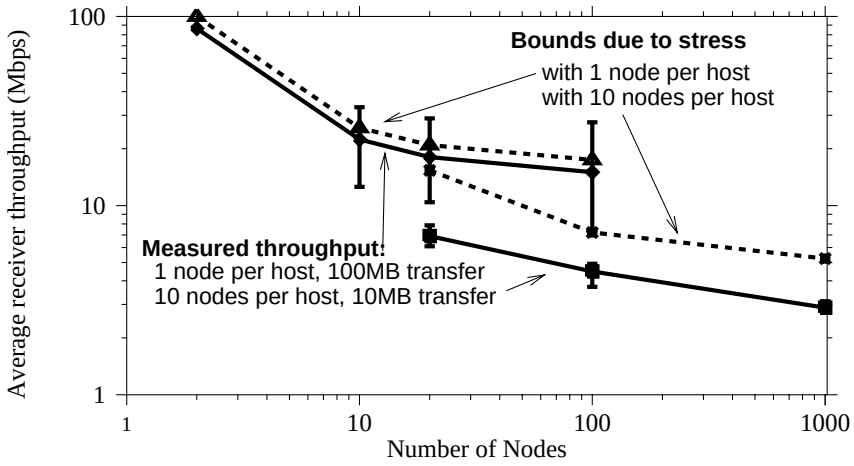


Figure 19: Measured multicast throughput at receivers. The plots compare theoretical bounds and measured values of the average throughput of multicast transmissions for a set of multicast trees. We show two measurements: (1) 100 MB bulk data transfers for an overlay with $N = 2 - 100$ nodes, where at most one node is assigned to each host; and (2) 10 MB bulk data transfers for an overlay with $N = 20 - 1000$ nodes, where always 10 nodes are assigned to each host. The error bars show the variance of the measurements across all considered multicast trees.

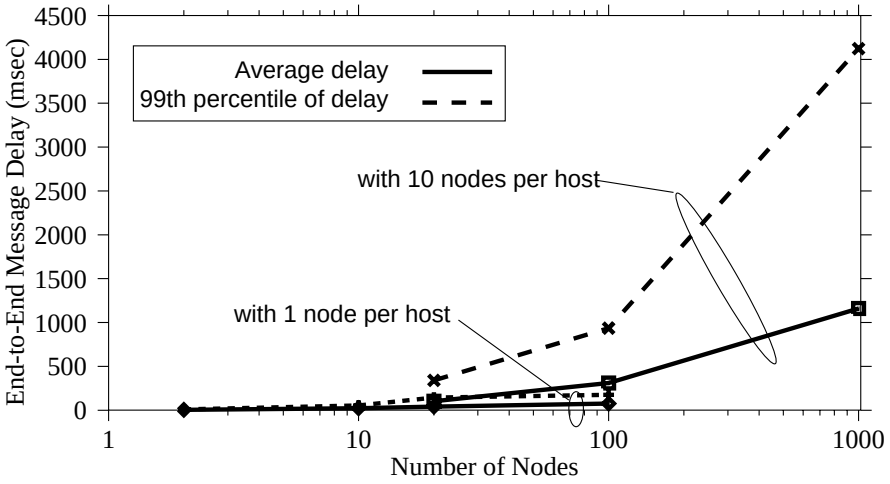


Figure 20: Measured end-to-end delay of messages. The plots show average delays and 99th percentile of delays for receivers in a set of multicast trees. We present two measurements: (1) 100 MB bulk data transfers for an overlay with $N = 2 - 100$ nodes, where at most one node is assigned to each host; and (2) 10 MB bulk data transfer for an overlay with $N = 20 - 1000$ nodes, where always 10 nodes are assigned to each host.

Figure 19 shows that the throughput for the multicast transmissions is high, achieving an average throughout of close to 15 Mbps when 100 nodes are running on 100 hosts. The throughput is lower when 10 nodes are run on each host. With 1 000 nodes running on 100 nodes, the measured throughput is close to 2 Mbps. Note that in the experiments which run one node per host, the achieved throughput is close to the theoretical bound. This indicates that the experiment is bandwidth limited in experiments where one overlay node is run on each host. When we run 10 overlay nodes on each host, the measured data is lower than the throughput bound. This indicates that the bottleneck is the processing at hosts.

Finally, we present measurements of the end-to-end delay experienced by individual messages during the above multicast experiments. Clocks on the hosts of the Centurion cluster are synchronized using *xntpd*, which runs the NTP Version 3 protocol [18], and should differ by at most 30 milliseconds. With the synchronized clocks, we can determine the end-to-end delay of a message for a node as the time lag between message transmission and message reception.

Figure 20 shows the average and 99th percentile values of the end-to-end packet delays for the multicast receivers for 10 MB bulk data transfers (with 10 nodes running on a host) and 100 MB bulk data transfers (with one node running on a host). If one node is run on a host, the average delay in a network with 100 nodes is 76 msec and the 99th percentile of the delay is 176 msec. For 100 nodes, using 10 nodes per host, the delay is much higher, with an average of 311 msec and a 99th percentile of the delay at 935 msec.

6 Conclusions

We have examined Delaunay triangulations as overlay topologies for application-layer multicast. We have presented a protocol, the *DT Protocol*, that creates and maintains a Delaunay triangulation overlay for applications.

The contribution of the presented Delaunay triangulation and the DT protocol is that we can build and maintain very large overlay networks with relatively low overhead, at the cost of poor resource utilization due to a possibly bad match of the overlay network to the network-layer infrastructure. To our knowledge, our study is the first that demonstrates in an implementation that overlay networks with 10 000 members can be built and maintained in a highly distributed fashion and with relatively low overhead.

There are several directions for future work. A limitation of this study is that the experiments with the DT protocol are confined to a local-area environment. We plan to evaluate the DT protocol over a wide-area network. However, managing wide-area measurement experiments for an overlay network with several thousand members imposes significant logistical problems. In addition, since Delaunay triangulations have multiple alternate paths between nodes, an overlay network based on Delaunay triangulations can mask a certain amount of node failures. The geographic routing techniques from [14] may be suitable to find routes to bypass failed nodes. Finally, we have pointed out that the mapping of the Delaunay triangulation overlay network to the network-layer infrastructure can be poor. New coordinate spaces, that can take into consideration network delay measurements, may provide an improved mapping of the logical overlay network to the underlying network.

As a final comment, the software of the DT protocol, which is part of the HyperCast software [10], will be made publicly available.

Acknowledgements

We thank Nicolas Christin, Haiyong Wang, Jianping Wang, and Guimin Zhang for their valuable feedback and comments.

References

[1] F. Baccelli, D. Kofman, and J.-L. Rougier. Self organizing hierarchical multicast trees and their optimization. In *Proceedings of IEEE Infocom '99*, pages 1081–1089, 1999.

[2] K. Calvert, M. Doar, and E. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[3] Y. Chawathe, S. McCanne, and E. A. Brewer. RMX: Reliable multicast for heterogeneous networks. In *Proceedings of IEEE Infocom*, pages 795–804, 2000.

[4] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, pages 1–12, 2000.

[5] J. Chuang. Economics of scalable network services. In *Proceedings of SPIE ITCOM 2001, Vol. 4526, Denver, CO*, pages 11–19, August 2001.

[6] Y. K. Dalal and R. M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, December 1978.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications*. Springer Verlag, 1997.

[8] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over a peer-to-peer network. Technical Report 2001-30, Stanford University (Computer Science Dept.), June 2001.

[9] P. Francis. Yoid: Extending the Internet multicast architecture, April 2000. http://www.aciri.org/yoid/docs/index.html.

[10] Multimedia Networks Group. Hypercast project. University of Virginia, http://www.cs.virginia.edu/~hypercast. 2001.

[11] D. Helder and S. Jamin. Banana tree protocol, an end-host multicast protocol. Technical Report CSETR-TR-429-00, University of Michigan, 2000.

[12] L. Hu. *Distributed Algorithms for Packet Radio Networks*. PhD thesis, University of California at Berkeley, 1990.

[13] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 197–212, San Diego, CA, October 2000. USENIX Association.

[14] B. N. Karp. *Geographic Routing for Wireless Networks*. PhD thesis, Harvard University, 2000.

[15] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry (CCCG'99)*, pages 51–54, Vancouver, August 1999.

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

[17] J. Liebeherr and B. S. Sethi. A Scalable Control Topology for Multicast Communications. In *Proceedings IEEE Infocom '98*, pages 1197–1203, 1998.

[18] D. Mills. The network time protocol (NTP) distribution. University of Delaware, http://www.eecis.udel.edu/~ntp/. 2001.

[19] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of 3rd Usenix Symposium on Internet Technologies and Systems*, March 2001.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001, San Diego*, pages 149–160, August 2001.

[21] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of Third International Workshop on Networked Group Communication (NGC '01), London, England*, 2001. To appear.

[22] V. Roca and A. El-Sayed. A host-based multicast (HBM) solution for group communications. In *1st IEEE International Conference on Networking (ICN'01)*, July 2001.

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

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001, San Diego*, pages 160–172, August 2001.

[25] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, April 2001.

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 **Leader With Neighbor** 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.
$\rightarrow$:	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	$\rightarrow$ **Leader Without Neighbor**

States: **Leader with Neighbor, Leader without Neighbor, Not Leader**	
Event	Action
Application exits	Send Goodbye to all neighbors Send Goodbye to server $\rightarrow$ **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 $t_{FastHeartbeat}$ 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 $t_{FastHeartbeat}$ ELSE Set timeout value of Heartbeat Timer to $t_{SlowHeartbeat}$

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 $t_{Backoff} = t_{FastHeartbeat}$ Send ServerRequest to alternate DT server ELSE Send ServerRequest to DT server Set $t_{Backoff} = 2\, t_{Backoff}$ Start Backoff Timer with a random time in $[0, t_{Backoff}]$
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 shifts its coordinates, i.e., $coord_{this}$ IF w passes neighbor test Add w as neighbor in neighborhood table Start Neighbor Timer for w with $t_{Neighbor}$ IF $coord_w > coord_{this}$ $\rightarrow$ **Not Leader** ELSE Set $t_{Backoff} = t_{FastHeartbeat}$ $\rightarrow$ **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 $t_{Backoff} = t_{FastHeartbeat}$ Send ServerRequest to alternate DT server ELSE Send ServerRequest to DT server Set $t_{Backoff} = 2\, t_{Backoff}$ Start Backoff Timer with a random time in $[0, t_{Backoff}]$
Receives ServerReply(w)	Set $t_{Backoff} = t_{FastHeartbeat}$ IF $w \neq$ this Send HelloNeighbor 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 $t_{FastHeartbeat}$ IF no neighbors in neighborhood table $\rightarrow$ **Leader without Neighbors**
HelloNeighbor or HelloNotNeighbor arrives from w	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 $t_{SlowHeartbeat}$ ELSE Set Heartbeat Timer to $t_{FastHeartbeat}$ 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 shifts its coordinates, i.e., $coord_{this}$ IF w passes the neighbor test Add w as neighbor 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, i.e., $coord_{this}$ While there exists a neighbor v with $coord_{this} = coord_v$ Node shifts its coordinates, i.e., $coord_{this}$ Remove all neighbors that fail neighborhood test ELSE IF message type is HelloNeighbor and w fails neighbor test Send HelloNotNeighbor to w IF there exists neighbor v with $coord_v > coord_{this}$ Clear Backoff Timer (if the timer was set) $\rightarrow$ **Not Leader**

State: **Not Leader**

Event	Action
Neighbor Timout for w or Goodbye arrives from w	IF w is a neighbor Remove w from neighbor table Set timeout value of Heartbeat Timer to $t_{FastHeartbeat}$ IF no neighbors left in neighborhood table Set $t_{Backoff} = t_{FastHeartbeat}$ Start Backoff Timer $\rightarrow$ **Leader Without Neighbor** ELSE IF $coord_{this} > coord_v$ for all neighbors v Set $t_{Backoff} = t_{FastHeartbeat}$ Start Backoff Timer $\rightarrow$ **Leader With Neighbor**
HelloNeighbor or HelloNotNeighbor arrives from w	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 $t_{SlowHeartbeat}$ ELSE Set Heartbeat Timer to $t_{FastHeartbeat}$ 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 shifts its coordinates, i.e., $coord_{this}$ IF w passes the neighbor test Add w as neighbor 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, i.e., $coord_{this}$ While there exists a neighbor v with $coord_{this} = coord_v$ Node shifts its coordinates, i.e., $coord_{this}$ Remove all neighbors that fail neighborhood test ELSE IF message type is HelloNeighbor and w fails neighbor test Send HelloNotNeighbor to w

State: **Leaving**

Event	Action
Application exits	$\rightarrow$ **Stopped**
Goodbye arrives from w	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 Start Leader Timer Send ServerReply(v) to node v $\rightarrow$ **Has Leader**

State: **Has Leader**

Event	Action
ServerRequest received from v	IF Leader $= v$ or v is in node cache IF v has changed its coordinates Update v's stored coordinates Re-select Leader ELSE IF $coord_v > coord_{Leader}$ Set Leader := v Start Leader Timer (since v is new Leader) IF node cache is full Remove one node cache entry Add 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 v ELSE Select w from node cache with $coord_w > coord_v$ Send ServerReply(w) to v IF Leader $\neq w$ and w has been used in 6 ServerReply messages 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 the largest coordinates ELSE IF Leader $= v$ and cache is empty $\rightarrow$ **Without Leader**
Heartbeat Timer expires	Send CachePing messages to every node in node cache
CachePong received from v	Restart Cache Entry timer for v

Input: byte array, denoted by "A[ ]".
Output: a 4-byte unsigned integer, denoted as "result".

Operators:
$Op1 \gg Op2$: $Op1$ is bit-wise right shifted $Op2$ times.
$Op1 \ll Op2$: $Op1$ 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 4: Procedure to compute the OverlayIDHash.

B Message Formats of the DT Protocol

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.

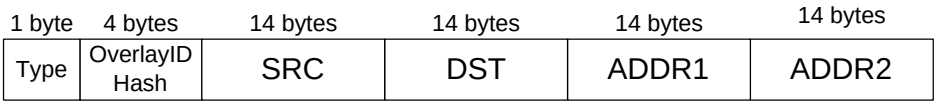


Figure 21: Message format of the DT protocol.

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 a variable-length byte array, is shown in Table 4.

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 Figure 22.

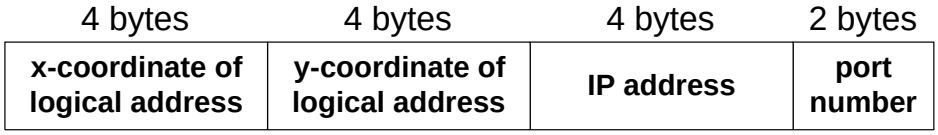


Figure 22: Format of a logical address/physical address.

- **HelloNeighbor/HelloNotNeighbor:** *SRC* and *DST* contain the addresses of the sending and receiving node[12] and the clockwise and counter-clockwise neighbors of the *ADDR1* and *ADDR2*, respectively, are the addresses 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.[13] 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 addresses, 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.

Remark: In the current implementation of the DT protocol, the physical address of the sender of a message of type HelloNeighbor, HelloNotNeighbor, or ServerRequest is included in the payload of the message. In Footnote 4, and other footnotes, we have pointed out that carrying the physical address of a sender in the

[12] See Footnote 7.
[13] See Footnote 8.

payload of a message may complicate the use of the DT protocol across a public/private network boundary. Instead, the receiver of a message should obtain the physical address of the sender of a message from the underlying transport protocol.