Efficient and Accurate Delaunay Triangulation Protocols under Churn *

Dong-Young Lee and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
{dylee, lam}@cs.utexas.edu

November 9, 2007
Technical Report # TR-07-59

Abstract

We design a new suite of protocols for a set of nodes in d-dimension ($d > 1$) to construct and maintain a distributed *Delaunay triangulation (DT) in a dynamic environment. The suite includes join, leave, failure, and maintenance protocols. The join, leave, and failure protocols are proved to be correct for a single join, leave, and failure, respectively. In practice, protocol processing of events may be concurrent. For a system under churn, it is impossible to maintain a correct distributed DT continually. We define an accuracy metric such that accuracy is 100% if and only if the distributed DT is correct. The maintenance protocol is designed to recover from incorrect system states and to improve accuracy. In designing the protocols in this paper, we made use of two new observations to substantially improve their efficiency. First, in the neighbor discovery process of a node, many replies to the node's queries contain redundant information. Second, the use of a new failure protocol that employs a* proactive *approach to recovery is better than the reactive approach used in [1, 2]. Experimental results show that our new suite of protocols maintains high accuracy for systems under churn and each system converges to 100% accuracy after churning stopped. They are much more efficient than protocols in prior work [1, 2].*

1. Introduction

Delaunay triangulation [3] and Voronoi diagram [4] have a long history and many applications in different fields of science and engineering including networking applications, such as greedy routing [5], finding the closest node to a given point, broadcast, multicast, etc.[1]. A triangulation for a given set S of nodes in a 2D space is a subdivision of the convex hull of nodes in S into non-overlapping triangles such that the vertexes of each triangle are nodes in S. A Delaunay triangulation in 2D is usually defined as a triangulation such that the circumcircle of each triangle does not include any node other than the vertexes of the triangle. Delaunay triangulation can be similarly generalized to a d-dimensional space[1] ($d > 1$) using simplexes instead of triangles [6].

We will use DT as abbreviation for "Delaunay triangulation." Our objective is a suite of protocols for a set of nodes in a d-dimensional space ($d > 1$) to construct and maintain a distributed DT. In designing these protocols, we allow the set of nodes to change with time. New nodes join the set (*system*) and existing nodes leave (gracefully) or fail[2] over time. The system is said to be *under churn* and the rate at which changes occur said to be the *churn rate*. We present in this paper a suite of four protocols for *join*, *leave*, *failure*, and *maintenance*.

In a distributed DT, each node maintains a set of its neighbors. By definition, a distributed DT of a set of nodes S is *correct* if and only if, for every node $u \in S$ on the distributed DT, u's neighbor set is the same as the set of u's neighbors on the (centralized) DT of S [1]. For convenience, we will sometimes say "the system state is correct" to mean "the distributed DT is correct." In a previous paper [1], we discovered and proved a *necessary and sufficient condition* for a distributed DT to be correct.

In designing the suite of protocols in this paper, we aim to achieve three properties: *correctness*, *accuracy*, and *efficiency*:

- Correctness – We prove the join, leave, and failure protocols to be correct for a single, join, leave, and failure, respectively. For the join protocol, we prove that if the system state is correct before a new node joins, and no other node joins, leaves, or fails during the join protocol execution, then the system state is correct after join pro-

*Research sponsored by National Science Foundation grant CNS-0434515.

[1] Delaunay triangulation is defined in a Euclidean space. When we say a d-dimensional space in this paper, we mean a d-dimensional Euclidean space.

[2] When a node fails, it becomes silent. We do not consider Byzantine failures.

tocol execution. A similar correctness property is proved for the leave and failure protocols. Note that these three protocols are adequate for a system whose churn rate is so low that joins, leaves, and failures occur *serially*, i.e., protocol execution finishes for each event (join, leave, or failure) before another event occurs. In general, for systems with a higher churn rate, we also provide a maintenance protocol, which is run periodically by each node.

- Accuracy – It is impossible to maintain a correct distributed DT continually for a system under churn. Note that correctness of a distributed DT is broken as soon as a join/leave/failure occurs and is recovered only after the join/leave/failure protocol finishes execution. Fortunately, some applications, such as greedy routing, can work well on a reasonably "accurate" distributed DT. We previously presented an accuracy metric for a distributed DT [1]; we will show that the accuracy of a distributed DT is 1 if and only if the distributed DT is correct. The maintenance protocol is designed to recover from incorrect system states due to concurrent protocol processing and to improve accuracy. We found that in all of our experiments conducted to date with the maintenance protocol, each system that had been under churn would converge to 100% accuracy some time after churning stopped.
- Efficiency – We use the total number of messages sent during protocol execution as the measure of efficiency. Protocols are said to be more efficient when their execution requires the use of fewer messages.

In a previous paper[1], we presented examples of networking applications to run on top of a distributed DT, namely: greedy routing, finding the closest existing node to a given point, clustering of network nodes, as well as broadcast and multicast withing a given radius without session states. Three DT protocols were presented: join and leave protocols that were proved correct and a maintenance protocol that was shown to converge to 100% accuracy after system churn. However, the join and maintenance protocols in this suite (the *old* protocols) were designed with correctness as the main goal and their execution requires the use of a large number of messages.

To make the *new* join and maintenance protocols in this paper much more efficient, we make two novel observations. First, the objective of the join protocols is for a new node n to identify its neighbors (on the global DT), and for n's neighbors to detect n's join. n sends a request to an existing node u for n's neighbors in u's local information. When n receives a reply, it learns new neighbors and sends requests to those newly-learned neighbors. This process is recursively repeated until u does not find any more new neighbor. Whereas it is necessary to send messages to all neighbors, since the neighbors need to be notified that n has joined, we discovered that n only needs to hear back from just one neighbor in each simplex that includes n rather than from all neighbors. Furthermore, queries as well as replies for some simplexes can be combined so that just one query-reply between n and one neighbor is needed for multiple simplexes. Based on this observation, we designed a new join protocol. We found that the new join protocol is much more efficient than our old join protocol. We have proved the new join protocol to be correct for a single join.

We also apply the above observation to substantially reduce the number of messages used by the new maintenance protocol. Furthermore, we make the following second observation to greatly reduce the total number of all protocol messages per unit time by reducing the frequency at which the new maintenance protocol runs.

We keep the leave protocol in [1] unchanged in this paper because it can efficiently address graceful node leaves. However, with the old suite of protocols, it is the old maintenance protocol's job to detect node failures and repair the resulting distributed DT. To detect a node failure, the node was probed by all of its neighbors. Furthermore, the distributed DT was repaired in a reactive fashion. The process of reactively repairing a distributed DT after a failure is inevitably costly, because the information needed for the repair was at the failed node and lost after failure.

To improve overall efficiency, we designed a new failure protocol to handle node failures. The failure protocol employs a *proactive* approach. Each node designates one of its neighbors as its *monitor node*. In the failure protocol, a node is probed only by its monitor node, eliminating duplicate probes. In addition, each node prepares a *contingency plan* and gives the contingency plan to its monitor node. The contingency plan includes all information to correctly update the distributed DT after its failure. Once the failure of a node is detected by its monitor node, the monitor node initiates failure recovery. That is, each neighbor of the failed node is notified of the failure as well as any new neighbor(s) that it should have after the failure. In this way, node failures are handled almost as efficiently as graceful node leaves in the leave protocol. We have proved the new failure protocol to be correct for a single failure.

Each node runs the maintenance protocol (new or old) periodically. The communication cost of the maintenance protocol increases as the period decreases (or frequency increases). Generally, as the churn rate increases, the maintenance protocol needs to be run more frequently. In the old protocol suite, moreover, the old maintenance protocol needs to be run at the probing frequency because one of its functions is to recover from node failures. With the inclusion of an efficient failure protocol in the new protocol suite to handle failures separately, the new maintenance protocol can be run less often. We found that the overall efficiency of the protocols as a whole is greatly improved as a result.

To the best of our knowledge, the only previous work for a dynamic distributed DT in a d-dimensional space is by Simon *et al.* [2]. They proposed two sets of distributed algorithms: basic generalized algorithms and improved generalized algorithms. Each set consists of an entity insertion (node join) algorithm and an entity deletion (node failure) algorithm. Their basic entity insertion algorithm is similar to our old join protocol. Their improved entity insertion algorithm is based on a centralized flip algorithm [7] whereas our join protocols are based on a "candidate-set approach" and our correctness condition for a distributed DT. The two approaches are fundamentally different. Their entity deletion algorithm and our failure protocols are also different. Our failure protocol is substantially more efficient than their improved entity deletion algorithm, which uses a reactive approach and allows duplicate probes. The centralized flip algorithm is known to be correct [8]. However, correctness of their distributed algorithms is not explicitly proved. Lastly, they do not have any algorithm, like our maintenance protocols, for recovery from concurrent processing of joins and failures due to system churn. As a result, their algorithms failed to converge to 100% accuracy after system churn in our simulation experiments.

A quick comparison of the four sets of protocols/algorithms is shown in Table 1. More detailed experimental results are presented in Section 7 of this paper.

	efficiency	convergence to 100% accuracy after system churn
Simon *et al.*'s basic algorithms	medium	No
Simon *et al.*'s improved algorithms	high	No
Our old protocols	low	Yes
Our new protocols	high	Yes

Table 1. A comparison of the old and new protocols with Simon *et al.*'s basic and improved algorithms.

The organization of this paper is as follows. In Section 2, we introduce the concepts and definitions of a distributed DT, present our system model, and a correctness condition for a distributed DT. We present the new join protocol in Section 3, the new failure protocol in Section 4, and the new maintenance protocol in Section 5. The accuracy metric is defined in Section 6 and experimental results are presented in Section 7. We conclude in Section 8.

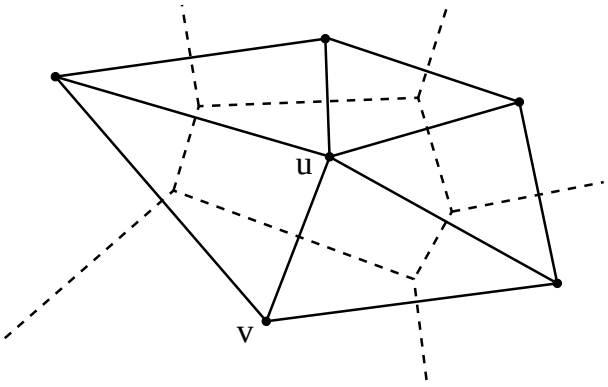

Figure 1. A Voronoi diagram (dashed lines) and the corresponding DT (solid lines) in a 2-dimensional space.

2. Distributed Delaunay triangulation

2.1. Concepts and definitions

Definition 1. *Consider a set of nodes S in a Euclidean space. The* **Voronoi diagram** *of S is a partitioning of the space into cells such that a node $u \in S$ is the closest node to all points within its Voronoi cell $VC_S(u)$.*

That is, $VC_S(u) = \{p \mid D(p,u) \leq D(p,w), \text{for any } w \in S\}$ where $D(x,y)$ denotes the distance between x and y. Note that a Voronoi cell in a d-dimensional space is a convex d-dimensional polytope enclosed by $(d-1)$-dimensional facets.

Definition 2. *Consider a set of nodes S in a Euclidean space. $VC_S(u)$ and $VC_S(v)$ are neighboring Voronoi cells, or* **neighbors** *of each other, if and only if $VC_S(u)$ and $VC_S(v)$ share a facet, which is denoted by $VF_S(u,v)$.*

Definition 3. *Consider a set of nodes S in a Euclidean space. The* **Delaunay triangulation** *of S is a graph on S where two nodes u and v in S have an edge between them if and only if $VC_S(u)$ and $VC_S(v)$ are neighbors of each other.*

Figure 1 shows a Voronoi diagram (dashed lines) for a set of nodes in a 2D space and a DT (solid lines) for the same set of nodes. $VC_S(u)$ and $VC_S(v)$ are neighbors of each other. We also say that u and v are neighbors of each other when $VC_S(u)$ and $VC_S(v)$ are neighbors of each other. Note that facets of a Voronoi cell perpendicularly bisect edges of a DT. Therefore, a DT is the dual of a Voronoi diagram.[3] Let us denote the DT of S as $DT(S)$.

Definition 4. *A* **distributed Delaunay triangulation** *of a set of nodes S is specified by $\{< u, N_u > \mid u \in S\}$, where N_u represents the set of u's neighbor nodes, which is locally determined by u.*

[3] In geometry, polyhedra are associated into pairs called duals, where the vertices of one correspond to the faces of the other.

Definition 5. *A distributed Delaunay triangulation of a set of nodes S is* **correct** *if and only if both of the following conditions hold for every pair of nodes $u, v \in S$: i) if there exists an edge between u and v on the global DT of S, then $v \in N_u$ and $u \in N_v$, and ii) if there does not exist an edge between u and v on the global DT of S, then $v \notin N_u$ and $u \notin N_v$.*

That is, a distributed DT is correct when for every node u, N_u is the same as the neighbors of u on $DT(S)$. Since u does not have global knowledge, it is not straightforward to achieve correctness.

2.2. System model

Our approach to construct a distributed DT is as follows. We assume that each node is associated with its coordinates in a d-dimensional Euclidean space. Each node has prior knowledge of its own coordinates, as is assumed in previous work [9, 10, 2, 11, 12]. The mechanism to obtain coordinates is beyond the scope of this study. Coordinates may be given by an application, a GPS device[13], or topology-aware virtual coordinates[14].[4] Also when we say a node u *knows* another node v, we assume that u knows v's coordinates as well.

Let S be a set of nodes to construct a distributed DT from. We will present protocols to enable each node $u \in S$ to get to know a set of its nearby nodes including u itself, denoted as C_u, to be referred to as u's *candidate set*. Then u determines the set of its neighbor nodes N_u by calculating a local DT of C_u, denoted by $DT(C_u)$. That is, $v \in N_u$ if and only if there exists an edge between u and v on $DT(C_u)$.

To simply protocol descriptions, we assume that message delivery is reliable. In a real implementation, additional mechanisms such as ARQ may be used to ensure reliable message delivery.

2.3. Correctness condition for a distributed Delaunay triangulation

Recall that a distributed DT is correct when for every node u, N_u is the same as the neighbors of u on $DT(S)$. Since N_u is the set of u's neighbor nodes on $DT(C_u)$ in our model, to achieve a correct distributed DT, the neighbors of u on $DT(C_u)$ must be the same as the neighbors of u on $DT(S)$. Note that C_u is local information of u while S is global knowledge. Therefore in designing our protocols, we need to ensure that C_u has enough information for u to correctly identify its global neighbors. If C_u is too limited, u cannot identify its global neighbors. For the extreme case of $C_u = S$, u can identify its neighbors on the global DT since $DT(C_u) = DT(S)$; however, the communication overhead for each node to acquire global knowledge would be extremely high.

[4] Application performance on a DT may be affected by the accuracy of virtual coordinates.

Theorem 1 (Correctness Condition)**.** *Let S be a set of nodes and for each node $u \in S$, u knows C_u, such that $u \in C_u \subset S$. The distributed DT of S is correct if and only if, for every $u \in S$, C_u includes all the neighbor nodes of u on $DT(S)$.*

Theorem 1, which was presented in our previous paper [1], identifies a *necessary and sufficient condition* for a distributed DT to be correct, namely: the candidate set of each node contains all of its global neighbors. In the following subsections, we use the above correctness condition as a guide to design our protocols. A proof of Theorem 1 is presented in [15].

3. Join protocols

3.1. Flip algorithm in a d-dimensional space

Flipping is a well-known and often-used technique to incrementally construct DT in 2D and 3D spaces. A centralized flip algorithm was also proposed to be used for a d-dimensional space [7] and was proved to be correct [8].

Note that two triangles in a 2D space are flipped into two other triangles, and two tetrahedra in a 3D space are flipped into three tetrahedra. In general, two simplexes in a d-dimensional space are flipped into d simplexes. This transformation is called 2-d flipping.

Incremental construction of DT based on flipping is as follows. When a new node is inserted, the simplex that encloses the new node is divided into $(d+1)$ new simplexes. Recall that the circum-hypersphere of a simplex on a DT should not include any other node except for the vertexes of the simplex. Each new simplex is checked whether its circum-hypersphere includes any other node. In case a simplex does include another node, it is flipped to generate new simplexes. The new simplexes are checked, and flipped if necessary. This process continues recursively. The flip algorithm requires a *general position assumption*, namely: no $d+1$ nodes are on the same hyperplane and no $d+2$ nodes are on the same hypersphere [6].

Figure 2 shows an example of flipping in a 2D space. A node n is inserted to a distributed DT. First, the simplex $\triangle uvw$ that encloses n is divided into three new simplexes (top figure). Then each new simplex is checked whether its circum-hypersphere includes any other node. In this example, the circum-hypersphere of $\triangle unv$ includes another node e. Therefore $\triangle unv$ and $\triangle uev$ are flipped into $\triangle une$ and $\triangle vne$ (bottom figure).

Distributed flip algorithms for joining were proposed for 2D[9], 3D[10], and a d-dimensional space [2]. The centralized flip algorithm is known to be correct (for a single join or serial joins). Since a simplex in d-dimension has $d+1$ nodes, operations at the $d+1$ nodes must be consistent in a distributed algorithm. Correctness has not been explicitly proved for any of the distributed algorithms.

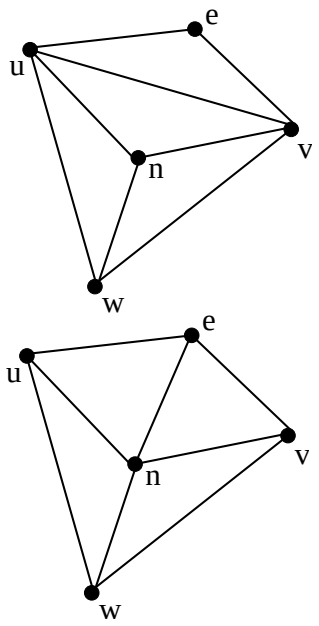

Figure 2. An example of flipping in 2D.

3.2. Candidate-set approach

In a previous paper [1], we proposed a join protocol based on the distributed system model using candidate sets and the correctness condition for a distributed DT introduced in Section 2. When a new node n joins a distributed DT, it is first led to the closest existing node z.[5] Then n sends a request to z for mutual neighbors of n and z on $DT(C_z)$. When n receives the reply, n puts the mutual neighbors in its candidate set (C_n) and re-calculates its neighbor set (N_n). If n finds any new neighbors, n sends requests to the new neighbors. This process is repeated recursively. We proved correctness of this protocol [15].

The flip algorithm and candidate-set approach are fundamentally different. However, it is interesting to note that there is a correspondence between the two. Table 2 shows how steps of the two different approaches correspond to each other.

Whereas the two approaches have corresponding steps, the steps are not exactly the same. For example, in step (b), n initially learns $(d+1)$ neighbors in the flip algorithm. In step (b) of the candidate-set approach, n may be informed of any nodes that z knows. In step (c) of the candidate-set approach, multiple neighbors may send duplicate messages to n to inform n of the same new neighbor. In step (c) of the flip algorithm, only one node may reply that a simplex is flipped. This observation gave us an idea to substantially improve the efficiency of our join protocol.

[5]This can be done using the protocol for finding a closest existing node in [1].

	Candidate-set approach	Flip algorithm
(a)	A joining node n is led to a closest existing node z.	A joining node n is led to a closest existing node.
(b)	z calculates local DT using C_z and n, and sends n's neighbors on $DT(C_z)$ to n.	The simplex that encloses n is divided into $(d+1)$ simplexes.
(c)	n contacts each of its new neighbors to see whether there are other potential neighbors.	The new simplexes are checked and flipped if necessary.
(d)	n recursively contacts new neighbors.	New (flipped) simplexes are recursively checked.

Table 2. Correspondence between join protocol in the candidate-set approach and flip algorithm.

3.3. New join protocol

Using the observation described above, we designed a new join protocol that is substantially more efficient than our old one. In addition to C_n and N_n, a joining node n keeps $N_n^{queried}$, which includes the neighbors that are already queried during its join process. Instead of querying all new neighbors, n queries only one neighbor for each simplex that does not include any node in $N_n^{queried}$. Note that only one neighbor in each simplex needs to be queried. If a simplex includes a node $v \in N_n^{queried}$, it means that the simplex has already been checked by v. Furthermore, queries as well as replies for multiple simplexes may be combined. The new join protocol requires the general position assumption, which was not required for the old join protocol.

Our new join protocol is still based on our candidate-set model and its correctness for a single join is proved using the correctness condition Theorem 1.

The new join protocol still has some duplicate computation. Even though there is only one query-reply interaction for each simplex, DT is calculated independently at each node.

Pseudocode of the new join protocol at a node is given in Figure 3. The protocol execution loop at a joining node, say n, and the response actions at an existing node, say v, are presented below.[6]

Protocol execution loop at a joining node n

At a joining node n, the join protocol runs as follows with a loop over steps 3-6:

[6]For clarity of presentation, the new join protocol is presented such that the joining node processes each NEIGHBOR_SET_REPLY message one at a time. We note that if the joining node can process multiple reply messages in step 3 of the loop, the number of query messages may be reduced; this change would not affect the correctness proof for the join protocol in the next section.

1. A joining node n is first led to a closest existing node z.
2. n sends a NEIGHBOR_SET_REQUEST message to z. C_n is set to $\{n, z\}$ and $N_n^{queried}$ is set to $\{z\}$.

Repeat steps 3-6 below until a reply has been received for every NEIGHBOR_SET_REQUEST message sent:

3. n receives a NEIGHBOR_SET_REPLY message from a node, say v. The message includes mutual neighbors of n and v on $DT(C_v)$.
4. n adds the newly learned neighbors (if any) to C_n, and calculates $DT(C_n)$.
5. Among simplexes that include n on $DT(C_n)$, simplexes that do not include any node in $N_n^{queried}$ are identified as unchecked simplexes. n selects some of its neighbors such that each unchecked simplex includes at least one selected neighbor.
6. n sends NEIGHBOR_SET_REQUEST messages to the selected neighbors. $N_n^{queried}$ is updated to include the selected neighbors. For the non-selected new neighbors, NEIGHBOR_NOTIFICATION messages are sent.

Response actions at an existing node v

- When v receives NEIGHBOR_SET_REQUEST from n, v puts n into C_v and re-calculates $DT(C_v)$. Then v sends to n NEIGHBOR_SET_REPLY that includes mutual neighbors of v and n on $DT(C_v)$.
- When v receives NEIGHBOR_NOTIFICATION from n, v includes n into C_v and re-calculates $DT(C_v)$. But v does not reply to n.

3.4. Correctness of the new join protocol

Lemma 1. *Let S be a set of nodes. For any subset C of S, let $u \in C$ and $v \in C$, if v is a neighbor of u on $DT(S)$, v is also a neighbor of u on $DT(C)$.*

Lemma 1 is proved in [15] (Lemma 3 in [15]).

Lemma 2. *Let n denote a new joining node, S be a set of existing nodes, and $S' = S \bigcup \{n\}$. Suppose that the existing distributed DT of S is correct and no other node joins, leaves, or fails. Let T be a simplex that exists on $DT(C_n)$ at some time during protocol execution and does not exists on $DT(S')$. Let $x \neq n$ be a node in T. Suppose that n sends a NEIGHBOR_SET_REQUEST to x. When n receives a NEIGHBOR_SET_REPLY from x, T is removed from $DT(C_n)$.*

Proof. Since the existing distributed DT of S is correct, C_x includes all the neighbors of x on $DT(S)$. After x receives NEIGHBOR_SET_REQUEST, C_x will include n, and thus C_x will include all the neighbors of x on $DT(S')$.

Consider the space that T occupies on $DT(C_n)$. Since T does not exist on $DT(S')$, the space is occupied by two or more different simplexes on $DT(S')$. Let T^* be one of those simplexes that includes both n and x. Note that there are $d-1$ other nodes in T^*, which are mutual neighbors of n and x on

Join(z) of node u
; Input: u is the joining node, if u is the only node in the system, $z = NULL$; otherwise z is the closest existing node to u.
if $z \neq NULL$ **then**
  Send(z, NEIGHBOR_SET_REQUEST)
  $C_u \leftarrow \{u, z\}$, $N_u \leftarrow \emptyset$, $N_u^{queried} \leftarrow \{z\}$
else
  $C_u \leftarrow \{u\}$, $N_u \leftarrow \emptyset$, $N_u^{queried} \leftarrow \emptyset$
end if

On u's receiving NEIGHBOR_SET_REQUEST from w
if $w \notin C_u$ **then**
  $C_u \leftarrow C_u \cup \{w\}$
  $N_u \leftarrow$ neighbor nodes of u on $DT(C_u)$
end if
$N_w^u \leftarrow \{x \mid x$ is a neighbor of w on $DT(C_u)\}$
Send(w, NEIGHBOR_SET_REPLY(N_w^u))

On u's receiving NEIGHBOR_SET_REPLY(N_u^w) from w
$C_u \leftarrow C_u \cup N_u^w$
Update_Neighbors(C_u, N_u)

On u's receiving NEIGHBOR_NOTIFICATION from w
if $w \notin C_u$ **then**
  $C_u \leftarrow C_u \cup \{w\}$
  $N_u \leftarrow$ neighbor nodes of u on $DT(C_u)$
end if

Update_Neighbors(C_u, N_u) of node u
$N_u^{old} \leftarrow N_u$
$N_u \leftarrow$ neighbor nodes of u on $DT(C_u)$
$N_u^{new} \leftarrow N_u - N_u^{old}$
$T_u^{new} \leftarrow$ simplexes that contain u on $DT(C_u)$ and do not contain any node in $N_u^{queried}$
$N_u^{check} \leftarrow Get\_Neighbors\_To\_Check(T_u^{new})$
for all $v \in N_u^{check}$ **do**
  Send(v, NEIGHBOR_SET_REQUEST)
end for
$N_u^{queried} \leftarrow N_u^{queried} \cup N_u^{check}$
$N_u^{notify} \leftarrow N_u^{new} - N_u^{check}$
for all $v \in N_u^{notify}$ **do**
  Send(v, NEIGHBOR_NOTIFICATION)
end for

Get_Neighbors_To_Check(T_u^{new}) of node u
$N'_u \leftarrow \emptyset$
while $T_u^{new} \neq \emptyset$ **do**
  $n \leftarrow$ a node in T_u^{new}
  $N'_u \leftarrow N'_u \cup n$
  remove each simplex that contains n from T_u^{new}
end while
Return $N'u$

Figure 3. New join protocol at a node u.

$DT(S')$, and, by Lemma 1, on $DT(C_x)$ as well. These $d-1$ nodes are included in the NEIGHBOR_SET_REPLY message from x to n. When n receives the NEIGHBOR_SET_REPLY message, the $d-1$ nodes are included in C_n and, by Lemma 1, become neighbors of n on $DT(C_n)$. As a result, T^* is created on $DT(C_n)$. That means T, which overlaps with T^*, is removed from $DT(C_n)$. □

Lemma 3. *Let n denote a new joining node, S be a set of existing nodes, and $S' = S \bigcup \{n\}$. Suppose that the existing distributed DT of S is correct and no other node joins, leaves, or fails. Then when the new join protocol finishes, C_n includes all the neighbor nodes of n on $DT(S')$.*

Proof. Consider a neighbor v of n on $DT(S')$. We show that v will be included in C_n when the join protocol finishes. At step 4 of the protocol execution loop, n has some nodes in C_n and calculates $DT(C_n)$.

Suppose that v is not yet included in C_n. Consider a straight line l from n to v. Let T be the first simplex on $DT(C_n)$ that l crosses. Such a simplex exists because v is not yet a neighbor of n on $DT(C_n)$. Note that T includes n. Let the other nodes of T be $x_1, x_2, ..., x_d$. Since l is an edge on $DT(S')$, T does not exist on $DT(S')$.

We show by contradiction that n has not received a NEIGHBOR_SET_REPLY message from any node in T. Suppose that n has received a NEIGHBOR_SET_REPLY from a node in T. Then by Lemma 2, T cannot exist on $DT(C_n)$, which contradicts the earlier assumption that T exists because v is not included in C_n yet.

Next we show that n will receive a NEIGHBOR_SET_REPLY message from a node in T. Either T includes a node x_a in $N_n^{queried}$ or T does not include any node in $N_n^{queried}$. In the former case, n has sent a NEIGHBOR_SET_REQUEST to x_a and will receive a NEIGHBOR_SET_REPLY message from x_a. In the latter case, by step 5 of the protocol execution loop, n will send a NEIGHBOR_SET_REQUEST to a node x_b in T and then receive a NEIGHBOR_SET_REPLY message from x_b. In each case, when n receives the NEIGHBOR_SET_REPLY message, T is removed from $DT(C_n)$ by Lemma 2.

Afterwards, if v is not a neighbor of n and l crosses another simplex on $DT(C_n)$, protocol execution continues and the above process repeats. This process finishes in a finite number of iterations since the number of nodes in S is finite and the number of possible simplexes in S is also finite. When there is no simplex that l crosses on $DT(C_n)$, l is an edge on $DT(C_n)$. Therefore v is included in C_n. □

The following theorem shows that our new join protocol is correct for a single join.

Theorem 2. *Let n denote a new joining node, S be a set of existing nodes, and $S' = S \bigcup \{n\}$. Suppose that the existing distributed DT of S is correct and no other node joins, leaves, or fails. Then when the new join protocol finishes, the updated distributed DT is correct.*

Proof. By Lemma 3, when the join process finishes, C_n will include all of its neighbor nodes on $DT(S')$. In addition, whenever n discovers a neighbor node v during the process, n sends either NEIGHBOR_SET_REQUEST or NEIGHBOR_NOTIFICATION message to v so that v includes n into C_v. Since the candidate sets of all existing nodes as well as the joining node are correctly updated, the updated distributed DT is correct by Theorem 1. □

4. Leave and failure protocols

In [1], we designed a leave protocol for nodes that leave gracefully. In the leave protocol, a leaving node u sends to each of its neighbors v a notification informing v of v's new neighbors after u leaves, as well as the fact that u is leaving. The leave protocol is very efficient. We keep it in our new protocol suite.

In this section, we propose a proactive approach to address node failures. Our failure protocol is almost as efficient as the leave protocol. It is proved to be correct for a single failure. The main idea is that every node u prepares a contingency plan in case it fails. That is, u calculates a local DT of only u's neighbors, not including itself. The contingency plan includes, for each neighbor v of u, new neighbor nodes of v after deleting u. u selects one of its neighbors, say m, and gives the contingency plan to m. m is called the *monitor node* of u. Then m periodically probes u to check whether u is alive. When m detects failure of u, m sends to each of u's former neighbors its portion of the the contingency plan. The protocol pseudocode is given in Figure 4.

The failure protocol takes over one of the functions of the old maintenance protocol. As a result, the new maintenance protocol may be run much less frequently, reducing overall cost of the system. As will be demonstrated by experiments for a system of nodes under churn, the new maintenance protocol is still necessary to recover from incorrect system states resulting from concurrent event occurrences.

Unlike the old maintenance protocol, probes are not duplicated in the failure protocol, since u is probed only by its monitor node. Furthermore, each former neighbor of u receives exactly 1 message upon u's failure. On the other hand, the failure protocol has the overhead of updating a contingency plan whenever a neighbor is added or deleted.

4.1. Correctness of the failure protocol

The following lemmas are proved in [15] (they correspond to Lemmas 4 and 10 in [15]).

Lemma 4. *Let S be a set of nodes. Consider $u \in S$ and a subset C_u of S that includes all the neighbor nodes of u on*

On change in N_u
$m_u \leftarrow$ the neighbor in N_u with the least ID
Calculate $DT(N_u)$; Note: $u \notin N_u$
for all $v \in N_u$ **do**
  $N_v^u \leftarrow \{w \mid w$ is a neighbor of v on $DT(N_u)\}$
end for
Send(m_u, CONTINGENCY_PLAN($\{< v, N_v^u > \mid v \in N_u\}$)

On u's receiving CONTINGENCY_PLAN(CP_v) from v
Set $FAILURE\_TIMER_v$ to $T + F$
; T is current time, F is the period of failure probe.

On u's expiration of $FAILURE\_TIMER_v$
Send(v, PING)
Set $PING\_TIMEOUT\_TIMER_v$ to $T + TO$
; T is current time, TO is the timeout value.

On u's receiving PING from v
if $v = m_u$ **then**
  Send(v, PONG($true$))
else
  Send(v, PONG($false$))
end if

On u's receiving PONG($flag$) from v
if $flag = true$ **then**
  Set $FAILURE\_TIMER_v$ to $T + F$
  ; T is current time, F is the period of failure probe.
else
  Cancel $FAILURE\_TIMER_v$
end if

On u's expiration of $PING\_TIMEOUT\_TIMER_v$
for all w that CP_v contains $< w, N_w^v >$ **do**
  Send(w, FAILURE_NOTIFICATION(v, N_w^v)
end for
$C_u \leftarrow C_u - \{v\} \cup N_u^v$
$N_u \leftarrow$ neighbor nodes of u on $DT(C_u)$
Cancel $FAILURE\_TIMER_v$

On u's receiving FAILURE_NOTIFICATION(v, N_u^v) from v
$C_u \leftarrow C_u - \{v\} \cup N_u^v$
$N_u \leftarrow$ neighbor nodes of u on $DT(C_u)$

Figure 4. Failure protocol at a node u.

$DT(S)$. If $v \in C_u$ is a neighbor of u on $DT(C_u)$, then v is also a neighbor of u on $DT(S)$.

Lemma 5. *Let S be a set of nodes and $S' = S - \{u\}$. Let v be a neighbor node of u on $DT(S)$. If w is a neighbor node of v on $DT(S')$, then w is a neighbor node of v on $DT(S)$ or w is a neighbor node of u on $DT(S)$.*

The following theorem shows that the failure protocol is correct for a single failure.

Theorem 3. *Let S be a set of nodes with a correct distributed DT. Suppose that a node $u \in S$ fails and its failure is detected by its monitor node $m \in S$, which then executes the failure protocol. Assume that there is no other join, leave, or failure. After the failure protocol finishes, the updated distributed DT is correct.*

Proof. Let $S' = S - \{u\}$. Consider a node $v \in S'$. The following case A shows that if v is not a neighbor of u, then v is not affected by the failure of u. Case B shows that if v is a neighbor of u, v will receive enough information from m to correctly update its candidate set.

Case A) Suppose that v is not a neighbor of u on $DT(S)$. Consider a node $w \in S', w \neq v$. If w is a neighbor of v on $DT(S')$, w is also a neighbor of v on $DT(S)$ by Lemma 1. If w is a neighbor of v on $DT(S)$, w is also a neighbor of v on $DT(S')$ by Lemma 4. Therefore the neighbors of v on $DT(S)$ are the same as the neighbors of v on $DT(S')$ and v is not affected by failure of u.

Case B) Suppose that v is a neighbor of u on $DT(S)$. Consider a node $w \in S', w \neq v$. If w is a neighbor of v on $DT(S')$, by Lemma 5, either w is already in C_v or w was a neighbor of u on DT(S). In the latter case, u's monitor node will notify v that w is its neighbor. Therefore C_v will include all the neighbor nodes of v on $DT(S')$.

From cases A and B, for each node $v \in S'$, C_v includes all the neighbor nodes of v on $DT(S')$. Therefore by Theorem 1, the updated distributed DT is correct. □

5. New maintenance protocol

The last member of our DT protocol suite is a new maintenance protocol. Even though the other protocols in the suite – the new join protocol, the (old) leave protocol, the new failure protocol – are proved to be correct for a single join, leave, and failure, respectively, nodes may join, leave, and fail concurrently for a system under churn. As to be shown by experimental results in Figure 9, neither our protocols (without a maintenance protocol) nor Simon *et al.*'s algorithms can recover a correct distributed DT after system churn. In that sense, our protocol suite is incomplete without a maintenance protocol, and so is Simon *et al.*'s set of insertion and deletion algorithms.

By Theorem 1, for a distributed DT to be correct, each node u must include in its neighbor set C_u all of its neighbor nodes on the global DT. This was one goal that our old maintenance protocol [1, 15] was designed to achieve. To that end, each node u periodically queries each of its neighbors to find any new neighbor of u that u is not aware of.

We found that running the maintenance protocol frequently requires a large communication cost. Note that the goal of a maintenance protocol is similar to that of a join protocol, namely, finding new neighbors. Therefore, we use the same technique as in the case of our new join protocol. That is, we reduce communication cost of our maintenance protocol by eliminating messages with redundant information.

Instead of querying all neighbors, a node u queries only one node for each simplex that includes u. Since a neighbor node may be included in multiple simplexes, the number of queried neighbors is much less than that of all neighbors.

Another goal of the old maintenance protocol was failure detection and recovery. In the old maintenance protocol, probing a node u was carried out by all neighbors of u. In our new set of protocols, the new failure protocol takes over the task of failure detection and recovery, where a node is probed by only one of its neighbor nodes. Thus the overall cost of our new maintenance and failure protocols is much less than the cost of the old maintenance protocol.

Although failure recovery is not a major goal of the new maintenance protocol, if a failure is detected by a message timeout, this information is propagated via DELETE messages. This may be necessary in case of concurrent failures. DELETE messages are propagated using the GRPB (greedy reverse-path broadcast) protocol in [1]. The maintenance protocol pseudocode (including GRPB) is given in Figure 5. Actions for receiving a NEIGHBOR_SET_REQUEST message and the functions of Update_Neighbors and Get_Neighbors_To_Check are the same as in Figure 3.

6. Accuracy metric for a system under churn

We define an accuracy metric as in [1], which we will use for experiments for a system of nodes under churn. We consider a node to be *in-system* from when it finishes joining to when it starts leaving. Let DDT_S be a distributed DT of a set of in-system nodes S. (Note that some nodes may be in the process of joining or leaving and not included.) Let $N_{correct}(DDT_S)$ be the number of correct neighbor entries of all nodes and $N_{wrong}(DDT_S)$ be the number of wrong neighbor entries of all nodes on DDT_S. A neighbor entry v of a node u is correct when v is a neighbor of u on the global DT (namely, $DT(S)$), and wrong when u and v are not neighbors on the global DT. Let $N(DT(S))$ be the number of edges on $DT(S)$. Note that edges on a global DT are undirectional and thus are counted twice when compared with neighbor entries. The accuracy of DDT_S is defined as follows:

$$accuracy(DDT_S) = \frac{N_{correct}(DDT_S) - N_{wrong}(DDT_S)}{2{\times}N(DT(S))}.$$

Observation 1. *The accuracy of a distributed DT is 1 if and only if the distributed DT is correct.*

Proof. (if) If the distributed DT is correct, $N_{correct}(DDT_S) = 2 \times N(DT(S))$ and $N_{wrong}(DDT_S) = 0$, resulting in accuracy of 1.

(only if) When accuracy is 1, we have $N_{correct}(DDT_S) - N_{wrong}(DDT_S) = 2{\times}N(DT(S))$. Since $N_{wrong}(DDT_S) \geq 0$, we get $N_{correct}(DDT_S) \geq 2{\times}N(DT(S))$. Also, $N_{correct}(DDT_S) \leq 2{\times}N(DT(S))$. It then follows that $N_{correct}(DDT_S) = 2{\times}N(DT(S))$ and $N_{wrong}(DDT_S) = 0$. That means the distributed DT is correct. □

On u's expiration of $PERIOD\_TIMER$
$N_u^{queried} \leftarrow \emptyset$
$T_u \leftarrow$ simplexes that contain u on $DT(N_u \cup \{u\})$
$N_u^{check} \leftarrow Get\_Neighbors\_To\_Check(T_u)$
for all $v \in N_u^{check}$ **do**
  Send(v, NEIGHBOR_SET_REQUEST)
  Set $NS\_TIMEOUT\_TIMER_v$ to $T + TO$
  ; T is current time, TO is the timeout value.
end for
Set $PERIOD\_TIMER$ to $T + P$
; T is current time, P is the period of failure probe.

On u's receiving NEIGHBOR_SET_REPLY(N_u^v) from v
$C_u \leftarrow C_u \cup N_u^v$
Update_Neighbors(C_u, N_u)
Cancel $NS\_TIMEOUT\_TIMER_v$

On u's expiration of $NS\_TIMEOUT\_TIMER_v$
$C_u \leftarrow C_u - \{v\}$
Update_Neighbors(C_u, N_u)
for all $w \in N_u$ **do**
  Send(w, DELETE(v, u))
end for

On u's receiving DELETE(v, w) from x
if $v \in C_u$ **then**
  $C_u \leftarrow C_u - \{v\}$
  Update_Neighbors(C_u, N_u)
  for all $y \in N_u$, $Dist(y, w) > Dist(u, w)$ **do**
    $N_{uy} \leftarrow$ nodes that share a simplex with both u and y on $DT(C_u)$
    if u is the closest node to w in N_{uy} **then**
      Send(y, DELETE(v, w))
    end if
  end for
end if

Figure 5. New maintenance protocol at a node u.

To demonstrate effectiveness of the new maintenance protocol, we designed an experiment for a system with an initial ring configuration. The system begins with a barely connected graph of 100 nodes, in which each node initially knows only one other node. That is, node p_i, $1 \leq i \leq 99$, initially has only p_{i-1} in its candidate set and its neighbor set; node p_0 knows p_{99}. Figure 6 shows change in accuracy of the distributed DT as the new maintenance protocol runs. The new maintenance protocol achieved a correct distributed DT within a few rounds of protocol execution.

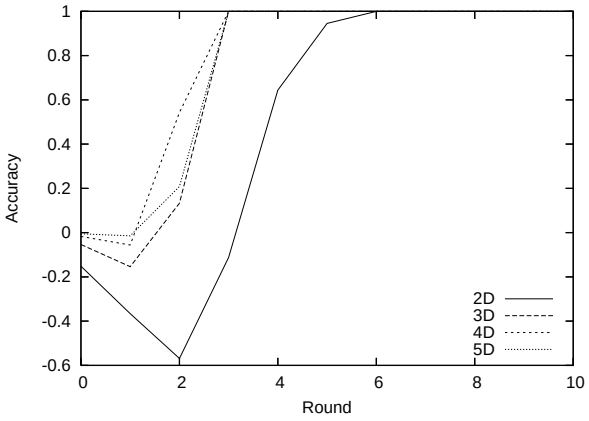

Figure 6. Accuracy of the maintenance protocol for a system with an initial ring configuration.

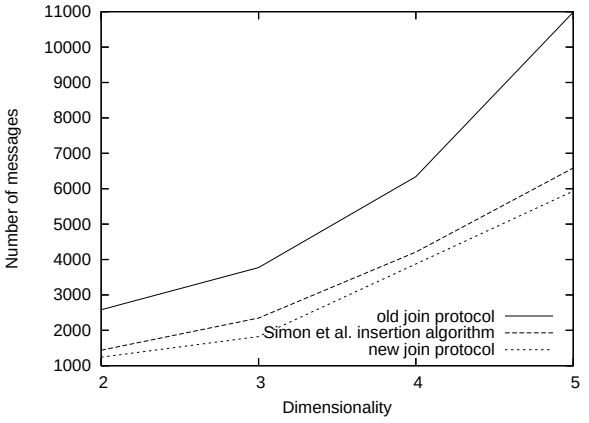

Figure 7. Costs of join protocols for 100 serial joins.

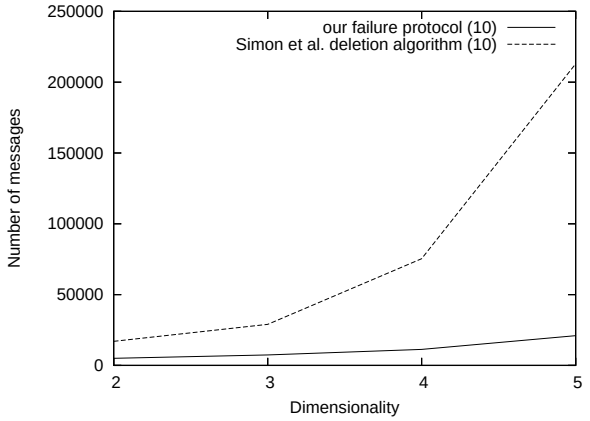

Figure 8. Costs of failure protocols for 100 serial failures.

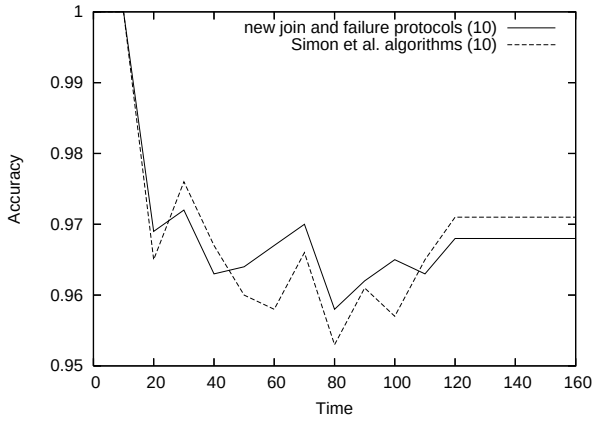

Figure 9. Accuracy without a maintenance protocol under system churn (join and fail).

7. Experimental results

7.1. Join protocols

Figure 7 shows communication costs of join protocols. Each curve shows the number of messages for 100 serial joins, increasing the system size from 200 nodes to 300 nodes, for different dimensionalities. Our new join protocol has much less cost than our old join protocol, and is slightly better than Simon *et al.*'s distributed algorithm.

7.2. Failure protocol

Figure 8 compares communication costs of our failure protocol and Simon *et al.*'s entity deletion algorithm. The number of messages used to recover from 100 serial failures from 300 initial nodes is measured. Both our failure protocol and Simon *et al.*'s deletion algorithm use the same probing period of 10 seconds. Our failure protocol is much more efficient than Simon *et al.*'s entity deletion algorithm.

7.3. Maintenance protocols

Figure 9 shows accuracy of our protocols without a maintenance protocol and Simon *et al.*'s algorithms under system churn. From a correct distributed DT of 400 initial nodes in 3D, 100 concurrent joins and 100 concurrent failures occur from time 10 to 110 second, with an average inter-arrival time of 1 second, respectively.[7] In both our failure protocol and Simon *et al.*'s entity deletion algorithm, nodes are probed every 10 seconds. The accuracy of the distributed DT is measured every 10 seconds. Both our new join and failure protocols and Simon *et al.*'s entity insertion and deletion algorithms cannot fully recover after system churn, resulting in an incorrect distributed DT.

[7] By Little's Law, for a system size of 400 nodes, the average lifetime of a node is 400 seconds. For P2P file sharing systems, this is a very high churn rate [16].

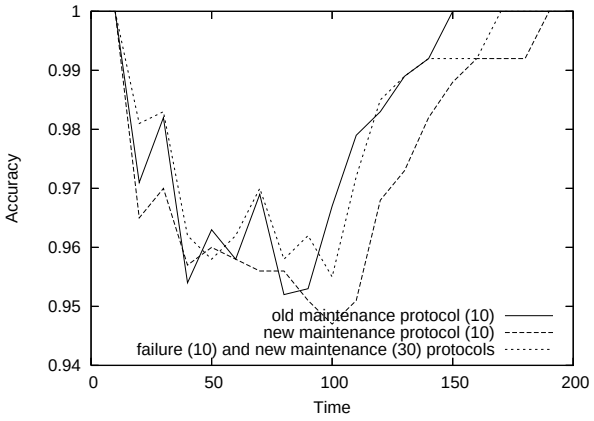

Figure 10. Accuracy of the old and new maintenance protocols under system churn (join and fail).

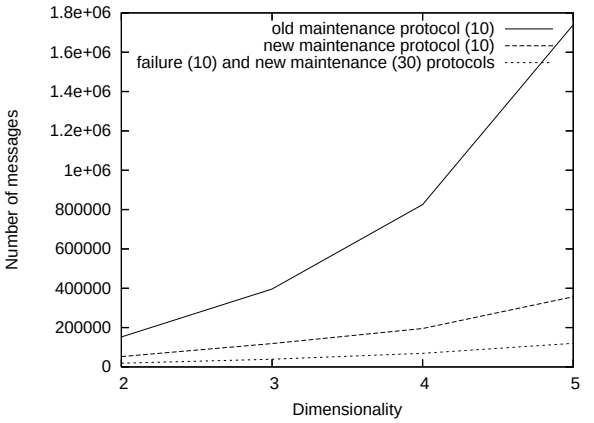

Figure 11. Costs of the old and new maintenance protocols under system churn (join and fail).

Figure 10 compares the accuracy of our protocols including the maintenance protocols in the same scenario as in Figure 9. The old maintenance protocol is run every 10 seconds. The new maintenance protocol is run every 10 seconds when it was run alone, and every 30 seconds when it was run along with the failure protocol which uses a probing period of 10 seconds. After system churn stops at time 110 second, accuracy converges to 100% by the maintenance protocols. The new maintenance protocol alone shows slightly lower accuracy than the old maintenance protocol. However, together with the failure protocol, the new maintenance protocol shows similar accuracy to that of the old maintenance protocol alone. The new maintenance protocol also took a longer time to converge to a correct distributed DT.

Figure 11 shows the communication costs in the above experiment. As expected, the new maintenance protocol has several times less communication cost compared to the old maintenance protocol. Efficiency is further improved when the new maintenance protocol is combined with the failure protocol, since the new maintenance protocol is run less frequently.

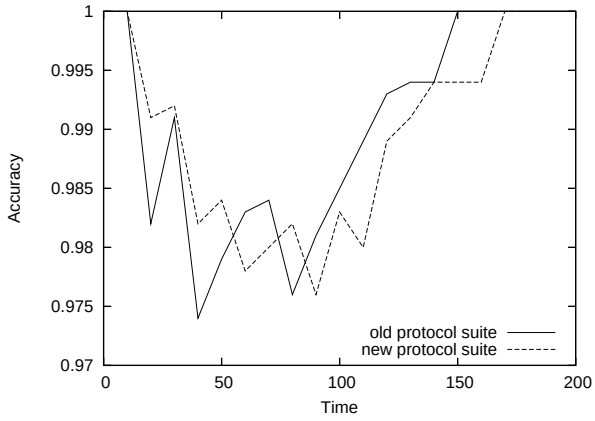

Figure 12. Accuracy of the old and new protocol suites under system churn (join, leave, and fail).

Figure 12 compares the accuracy of our old and new protocol suites under system churn, where nodes join, leave, and fail concurrently. The scenario is similar to that of the previous experiment, except that nodes either gracefully leave or fail instead of all failing. From a correct distributed DT of 400 initial nodes, 100 joins, 50 leaves, and 50 failures occur from time 10 to 110. The average inter-arrival time of joins is 1 second and those of leaves and failures are 2 seconds, respectively. The old maintenance protocol is run every 10 seconds. The new maintenance protocol is run every 30 seconds along with the failure protocol which uses a probing period of 10 seconds. The new protocol suite maintains about the same accuracy as the old protocol suite. In the end, the new protocol suite took a slightly longer time to converge to a correct distributed DT, because it uses a longer period (30 seconds instead of 10 seconds).

Figure 13 shows the communication costs of our old and new protocol suites in the same churn experiment. The new protocol suite has many times less communication cost compared to the old protocol suite.

8. Conclusions

While DT has been known and used for a long time in different fields of science and engineering, the design of protocols for constructing and maintaining a distributed DT for a dynamic system has not received much attention. In a previous paper [1], we investigated the design of several application-level protocols to support DT applications on networks, as well as join, leave, and maintenance protocols to construct and maintain a distributed DT. Our focus therein

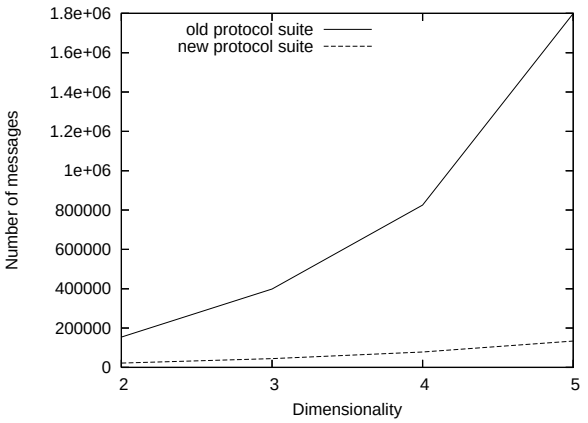

Figure 13. Costs of the old and new protocol suites under system churn (join, leave, and fail).

was to ensure the correctness of a distributed DT. Towards that goal, we defined a correct distributed DT, discovered a necessary and sufficient condition for a distributed DT to be correct, and defined an accuracy metric. Our old join and leave protocols were proved to be correct for a single join and leave, respectively. Our old maintenance protocol was found to converge to a correct distributed DT with 100% accuracy after system churn had stopped.

Efficiency, however, was not our primary protocol design goal in [1]. In this paper, we design a new suite of DT protocols that are correct, accurate, and efficient. The new protocols in this paper are vastly more efficient as a result of two new ideas in this paper. First, in the neighbor discovery process of a node, say n, it needs to hear back from just one neighbor in each simplex that includes n rather than from all neighbors. Furthermore, queries as well as replies for some simplexes can be combined in a single query-reply. We made use of this idea to greatly improve the efficiency of the new join and maintenance protocols. Second, we have added an efficient failure protocol that employs a proactive approach to failure recovery, instead of relying on the maintenance protocol which recovers from failures reactively. With addition of the failure protocol, the maintenance protocol can be run less often and the overall system efficiency improves. Both the new join and failure protocols are proved to be correct for a single join and failure, respectively. The old leave protocol was highly efficient and it is kept in the new protocol suite. Experimental results show that our new suite of protocols maintains high accuracy for systems under churn and each system converges to 100% accuracy after churning stopped.

Acknowledgement

We thank Professor Paul A. G. Sivilotti at the Ohio State University for motivating this study. Correspondence between the join protocol based on candidate sets and the flip algorithm was found during discussions with him.

References

[1] Dong-Young Lee and Simon S. Lam, "Protocol Design for Dynamic Delaunay Triangulation," *Proceedings of IEEE ICDCS 2007,* Toronto, Canada, July 2007.

[2] Gwendal Simon, Moritz Steiner, Ernst Biersack, "Distributed Dynamic Delaunay Triangulation in d-Dimensional Spaces," Technical report, Institut Eurecom, August 2005.

[3] B. Delaunay, "Sur la sphère vide," *Otdelenie Matematicheskikh i Estestvennykh Nauk,* 7:793-800, 1934.

[4] G. Voronoï, "Nouvelles applications des paramètres continus à la théorie des formes quandratiques," *Journal für die Reine and Angewandte Mathematik,* 133:97-178, 1908.

[5] Prosenjut Bose and Pat Morin, "Online routing in triangulations," *SIAM Journal on Computing,* 33(4):937-951, 2004.

[6] Edited by P. M. Gruber and J. M. Wills, *Handbook of Convex Geometry,* North-Holland, 1993.

[7] D. F. Watson, "Computing the n-dimensional Delaunay tesselation with application to Voronoi Polytopes," *The Computer Journal,* 24(2):167–172, 1981.

[8] H. Edelsbrunner and N. R. Shah, "Incremental topological flipping works for regular triangulation," *Algorithmica,* 15:223–241, 1996.

[9] Jörg Liebeherr, Michael Nahas, "Application-Layer Multicasting With Delaunay triangulation Overlays," *IEEE Journal on Selected Areas in Communications,* VOL. 20, NO. 8, October 2002.

[10] Moritz Steiner, Ernst Biersack, "A fully distributed peer to peer structure based on 3D Delaunay Triangulation," *Proceedings of Algotel 2005*.

[11] Masaaki Ohnishi, Ryo Nishide, Shinichi Ueshima, "Incremental Construction of Delaunay Overlaid Network for Virtual Collaborative Space," *Proceedings of the third international conference on creating, connecting and collaborating through computing,* 2005.

[12] Taewon Yoo, Hyonik Lee, Jinwon Lee, Sunghee Choi, Junehwa Song, "Distributed Kinetic Delaunay Triangulation," CS/TR-2005-240, KAIST, Korea, 2005.

[13] Tommaso Melodia, Dario Pompili, Ian F. Akyildiz, "A Communication Architecture for Mobile Wireless Sensor and Actor Networks," *Proceedings of IEEE SECON 2006,* September 2006.

[14] T. S. Eugene Ng and Hui Zhang, "Predicting Internet Network Distance with Coordinates-Based Approaches," *Proceedings of IEEE Infocom 2002,* June 2002.

[15] Dong-Young Lee and Simon S. Lam, "Protocol design for dynamic Delaunay triangulation," The Univ. of Texas at Austin, Dept. of Computer Sciences, Technical Report TR-06-48, Oct 2006.

[16] S. Sariou, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," *Proceedings of Multimedia Computing and Networking,* January 2002.