Distributed Dynamic Delaunay Triangulation in d-Dimensional Spaces

Gwendal Simon[•] Moritz Steiner[⋆◇] Ernst Biersack[◇]

• France Telecom - R&D Division
38 rue du Général Leclerc
92794 Issy-Moulineaux Cedex
France

◇ Institut Eurécom
2229 route des Crêtes
06904 Sophia-Antipolis
France

⋆ University of Mannheim
Computer Science IV - A5, 6
68159 Mannheim
Germany

moritz.steiner@informatik.uni-mannheim.de
Tel: +49 621 181 2614 — Fax: +49 621 181 2601

Abstract

Voronoi diagrams and Delaunay triangulations have proved to be efficient solutions to numerous theoretical problems. They appear as an appealing structure for distributed overlay networks when entities are characterized by a position in a d-dimensional space. In this paper, we present some algorithms aiming to maintain an overlay network matching the Delaunay triangulation of the participating entities. We consider that entities are characterized by a position in a d-dimensional space. We first study the insertion of a new entity, then we present the deletion of an entity.

Keywords: Distributed Computing, Computational Geometry, Peer-to-Peer Systems

1 Introduction

Study Context In shared virtual reality applications, *entities* are characterized by a position in a virtual space. As entities interact according to their virtual proximity, the application should ensure that each entity is aware of all entities within its virtual surrounding. Some fully decentralized systems [1, 12, 13] intend to create two-dimensional scalable virtual worlds through an overlay network specifically designed to achieve this end.

Building an overlay network based on a Delaunay triangulation [12] seems a convincing approach. As depicted in Figure 1, a Delaunay triangulation [7] links two entities when their Voronoi regions share a boundary [3, 19]. A Voronoi diagram [23] of a set of entities in an euclidean space tessellates the whole space. So, any subspace S of a virtual world can be monitored by the entities whose Voronoi region overlaps S. Hence, an entity e can be notified of any intrusion within its *area of interest* $S(e)$ if it knows all entities monitoring $S(e)$ [12].

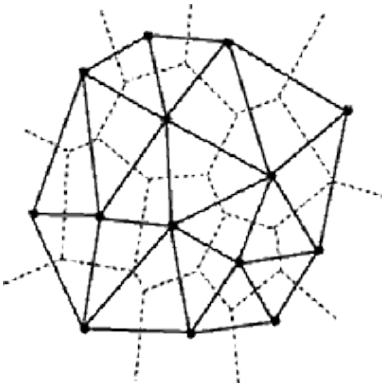

Figure 1: Delaunay triangulation and Voronoi diagram (dashed lines) in two-dimensional space

We initially purpose to conceive a *three-dimensional* virtual reality system based on a distributed Delaunay-based overlay network.

Related Motivations In network positioning systems [6, 18, 20], each host is characterized by a position in a $d-$dimensional space. In most cases, it is possible to predict the network distance between two hosts by computing the distance between their

respective positions.

A claimed application is an overlay application-layer multicast infrastructure for multimedia streaming systems [25]. The idea is to build a multicast tree linking preferentially hosts that are close in the space. As the euclidean minimum spanning tree is a subgraph of a Delaunay triangulation [21], a Delaunay-based overlay network could be an appealing structure for such distributed systems.

Moreover, relevant cluster structures are well reflected by the Voronoi diagrams, which provide efficient solutions to both partitional and hierarchical clustering [14]. These properties could be used by latency-based distributed content delivery networks [22]. The structure could ease not only the detection of a group of close hosts, but also the selection of an accurate place for replicas.

Our secondary purpose is to conceive a distributed Delaunay-based overlay structure from network positions in *d-dimensional spaces*.

Contributions Previous studies on Delaunay-based overlay in dynamic distributed systems [2, 16] rely on an angular feature which is specific to two-dimensional spaces. In [15], an overlay based on a Delaunay triangulation is constructed on a ad-hoc network, but some features of wireless protocols ease the detection of neighbors and the algorithms focus on two-dimensional space. Therefore, these algorithms can not be applied to higher dimensional spaces.

We design an overlay network matching the Delaunay triangulation of the participating entities in a d-dimensional space. The Section 3 is devoted to the description of the self-organizing algorithm for entity insertion. The entity deletion algorithm is presented in Section 4.

2 Model and Definitions

An *entity* is a process having communication capabilities and running on a end user's computer. We assume that unpredictable failures are detected in a reasonable time. The entities are able to exchange messages through reliable bidirectional communication links. Each entity is characterized by a *position* in a $d-$dimensional space and a unique network identifier allowing any other process to contact it. A connection occurs between two entities when both entities store their respective network identifier. The set of entities connected to an entity e at time t is denoted $K(e,t)$. The system at time t is modeled by a graph $G(t) = (V(t), E(t))$ where $V(t)$ is the set of entities and $E(t)$ is the set of connections.

The Delaunay triangulation of $V(t)$, noted $DT(t)$, links entities into non-overlapping $d-$simplices such that the circum-hypersphere of each $d-$simplex contains none of the entities in its interior. We would like to ensure that $\forall t, E(t) = DT(t)$.

The hypersphere which passes through all entities of a $d-$simplex T is noted $\mathcal{C}(T)$. An entity z is inside $\mathcal{C}(e_o, e_1 \dots e_d)$ if:

$$\begin{vmatrix} e_0[1] & \dots & e_0[d] & e_0[1]^2 + \dots + e_0[d]^2 & 1 \\ e_1[1] & \dots & e_1[d] & e_1[1]^2 + \dots + e_1[d]^2 & 1 \\ \vdots & & \vdots & & \vdots \\ e_d[1] & \dots & e_d[d] & e_d[1]^2 + \dots + e_d[d]^2 & 1 \\ z[1] & \dots & z[d] & z[1]^2 + \dots + z[d]^2 & 1 \end{vmatrix} > 0$$

We assume in this paper that entities are in *general position*, *i.e.* no $d+1$ entities are on the same hyperplane and no $d+2$ entities are on the same hypersphere. Finally, we consider that the position chosen by a new entity at t is in the interior of the convex hull of $V(t)$.

3 Entity Insertion

In the following, we first describe a very simplistic distributed version of the generalized entity insertion algorithm. This straightforward algorithm requires each neighbor of the new entity to recompute the new triangulation. Then, we propose an enhanced algorithm aiming to reduce the overall computation cost.

3.1 Basic Generalized Algorithm

We consider a new entity z joining the system at time t. We assume that z has a position in the space and knows at least one entity in $V(t)$.

The entity z should first discover the entity $w \in V(t)$ such that w is the closest entity to the position of z. The detection of w can be achieved by a *greedy walk* initiated by contacting any entity in $V(t)$. A distributed iterative implementation requires two distinct messages: a message `find-nearest`, sent by z, requesting the destination's closest neighbor to the position of z, and a message `nearest`, sent to z, containing the identifier of the emitter's closest neighbor to z. A greedy walk algorithm is known to always succeed in a Delaunay triangulation [5].

The entity w is connected with a set of d entities $V_z \subseteq K(w,t)$ such that $V_z \cup \{w\}$ generates a $d-$simplex enclosing z. These entities are the closest neighbors of z in the new Delaunay triangulation. From them, a recursive process allows z to discover and contact all of its neighbors in $DT(t)$. The principle is as follows: z sends a message `hello` to each new neighbor. They answer by a message `detect` containing either some new neighbors, either nothing. The process ends when all contacted neighbors have answered.

We consider an entity a receiving a message `hello` from z at time t. The entity a should re-construct a new Delaunay triangulation for $K(a,t) \cup \{a,z\}$, noted $LDT(a,t)$. Any known algorithm can be used [3, 19]. Then, the entity a sends to z a message `detect` containing all entities $e \in K(a,t)$ such that (e,z) exists in $LDT(a,t)$.

Upon reception of a message `detect`, the entity z first adds the unknown new neighbors to its local Delaunay triangulation $LDT(z,t)$. These new neighbors may reveal that some contacted entities are not neighbors in Delaunay. So, some message `close` are sent to them. Then, z sends a message `hello` to its not yet contacted neighbors in $LDT(z,t)$. Finally, the entity z stores the emitter of the message `detect` in $K(z,t)$.

This distributed algorithm naturally leads to an overlay matching the Delaunay triangulation. Moreover, as a Delaunay triangulation is unique if entities are in general position, simultaneous insertions may only cause short local incoherency. Indeed, the order of message arrival has no consequences on the overlay structure.

Unfortunately, this algorithm admits huge computation costs. Each neighbor of the new entity should re-compute the local Delaunay triangulation. This task requires numerous unnecessary computations that are not suitable, especially in a very dynamic virtual world. Moreover, an entity $e \in K(z,t)$ is detected by all entities in $K(e,t) \cap K(z,t)$, so appears in a large number of messages `detect`. Such redundancy is unnecessary.

3.2 Improved Generalized Algorithm

In two-dimensional spaces, a known technique depicted in Figure 2 consists of finding the triangle enclosing the new entity, then splitting this triangle into three, finally recursively checking on all adjacent triangles whether the *edge flipping* procedure should be applied [10, 11]. In Figure 3, the edge flipping algorithm replaces the edge (b,c) by the edge (a,z) because $\mathcal{C}(a,b,c)$ contains the new entity z.

Few papers study the behavior of the flipping mechanism in $d-$dimensional space. Most notably, an incremental algorithm for the triangulation construction is proposed in [24] and, at a later time, the flipping mechanism has been proved to always succeed in constructing the triangulation [9].

We propose a distributed algorithm inspired by the edge flipping mechanism. It mainly bases on geometrical objects. An entity e stores at time t in a personal buffer $\mathcal{T}(e,t)$ the description of all $d-$simplices in $DT(t)$ it is involved in. A description of a $d-$simplex T is a list of the network identifier and the position of all entities generating T.

The algorithm exhibits three rounds. The first one aims to discover the enclosing d-simplex. The greedy walk detailed in Section 3.1 can be used even if it requires $\mathcal{O}(n)$ time in the worst case and $\mathcal{O}(n^{1/d})$ expected time. Unfortunately, this issue is not yet resolved. The following details the two latter rounds for the insertion of a new entity z.

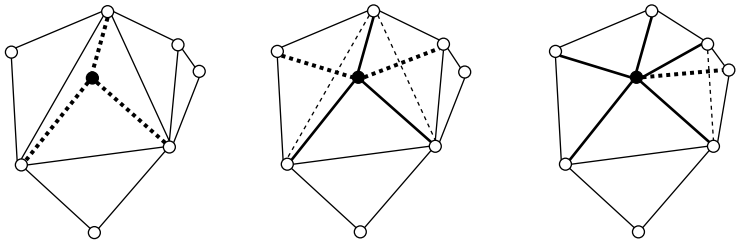

Figure 2: insertion of a new entity

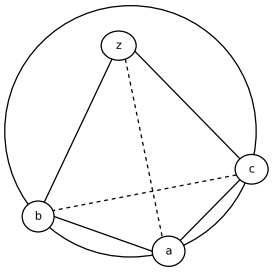

Figure 3: edge flipping

3.2.1 Splitting the Enclosing Simplex

An entity enclosed in a $d-$simplex splits it into $d+1$ $d-$simplices. For instance, a $2-$simplex, namely triangle, is split into three triangles as illustrated in Figure 2. In Figure 4, a new entity z belonging to a tetrahedron $T = (a, b, c, d)$ splits it into four tetrahedra $T_0 = (a, b, c, z)$, $T_1 = (a, b, d, z)$, $T_2 = (a, c, d, z)$ and $T_3 = (b, c, d, z)$.

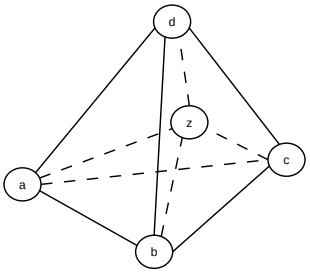

Figure 4: splitting the enclosing tetrahedron

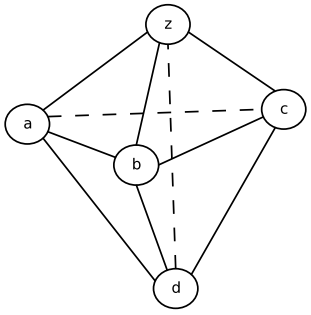

Figure 5: a 2-3 flip

The first round ends when the entity z receives a description of the $d-$simplex T enclosing its position. The entity z splits T into $d+1$ non-overlapping $d-$simplices. Then, it stores these simplices in $\mathcal{T}(z, t)$. Finally, z sends a message `hello` to its $d+1$ new neighbors. This message contains the former d-simplex T and the d new simplices in which the destination is involved. In the example of Figure 4, the entity z sends to its neighbor b a message `hello` containing T, T_0, T_1 and T_3.

3.2.2 Recursive Flipping Mechanism

We consider an entity a receiving a message `hello` from the new entity z at time t. This message contains a $d-$simplex T to be discarded and d new simplices T_1, $T_2 \ldots T_d$ containing both a and z.

In the first case, the simplex T does not exist in $\mathcal{T}(a, t)$. This unusual situation may be due to communication latencies or simultaneous events. The simplex T has been discarded because one entity $b \in K(a, t)$ is within $\mathcal{C}(T)$. The entity a should detect this entity and informs z that T should not be considered as a valid $d-$simplex. The detection may be eased by maintaining a dedicated structure called *Delaunay tree* [4]. Upon reception of this message, the new entity z should cancel its previous actions and resume the first round.

In a normal execution, the triangulation is updated by recursively performing a flipping mechanism that splits two $d-$simplices into d $d-$simplices. This operation is called $2-d$ flip. In Figure 5, two tetrahedra (a, b, c, z) and (a, b, c, d) result in three tetrahedra (a, b, z, d), (a, c, z, d) and (b, c, z, d). For simplicity, we restrict here the study to one simplex T_1 among the d simplices contained in the message `hello` received by a.

The entity a should first determine the $d-$simplex $T_1' \in \mathcal{T}(a, t)$ such that T_1' and T_1 share one common $(d-1)$-simplex, *e.g.* $T_1 = (a, e_0, \ldots, e_{d-2}, z)$ and $T_1' = (a, e_0, \ldots, e_{d-2}, e)$. The entity e may be considered as the opposite of z through this $(d-1)-$simplex. For instance, in two dimensions, e is the opposite of z through the edge (a, e_0) and, in three dimensions, e is the opposite of z through the face (a, e_0, e_1).

If z belongs to the circum-hypersphere of T_1', then the entity a should immediately inform z that (1) the $d-$simplex T_1 should be discarded and (2) e should be considered as a new neighbor. This is achieved by a message `detect` containing both a description of e and T_1. Then, the entity a operates

the $2-d$ flip, resulting in d $d-$simplices, noted T_{11}, T_{12}, ..., T_{1d}. One of these $d-$simplices does not contain a but all other should be inserted in $\mathcal{T}(a,t)$. Meanwhile, T_1 and T_1' are discarded.

The operation described above should be reiterated with the newly created simplices. For instance, if we consider T_{11}, the entity a should first look for the $d-$simplex $T_{11}' \in \mathcal{T}(a,t)$ sharing a $(d-1)-$simplex with T_{11}. Then, a should verify whether the new entity z belongs to $\mathcal{C}(T_{11}')$. If so, a splits T_{11} and T_{11}' into d $d-$simplices and sends another `detect` message to z.

This process ends until a does not split any new $d-$simplex. The entity a should then disconnect from the entities with which it does not share any $d-$simplex in $\mathcal{T}(a,t)$.

In this algorithm, all entities involved in a invalid simplex T should perform the edge flipping mechanism and send a message `detect` to z. However, if we consider reliable communication links, this message redundancy is unnecessary. It is possible to conceive a function `select` that can be executed by the d entities involved in T and which returns `true` for only one entity. This entity is the only one to send a message `detect` to z. For instance, the function `select` may be implemented by comparing the distance with z. Note that, in the algorithm of Section 3.1, the way to select one entity is harder as no entity knows whether its neighbors have been yet contacted by the new entity while it is sure here that all entities involved in a simplex receives a message `detect`.

Algorithm 1 shows the pseudocode of the treatment at reception of a `hello` message. The first test relates to inconsistency detection. The function `detectInside` returns an entity b within $\mathcal{C}(T)$ (lines 2-3). In other cases, the entity receives the notification of d new $d-$simplices $T_1 \dots T_d$. It puts them on a queue Q managed by a first-in-first-out policy (lines 5-6). Then, it retrieves the $d-$simplex T_a (line 8). We consider a function `share` which takes in argument T_a and returns a $d-$simplex and an entity such that the $d-$simplex shares with T_a a $(d-1)-$simplex and the entity is the opposite of z through this $(d-1)-$simplex (line 9). If the *in-*

Algorithm 1: `hello z` T T_1 ... T_d

```
1  if T ∉ T(a,t) then
2      b ← detectInside (T)
3      send "cancel b" to z
4  else
5      for i = 1...d do
6          Q.put (T_i)
7      while Q ≠ ∅ do
8          T_a ← Q.pop ()
9          T_b, e = share (T_a)
10         if z ∈ C(T_b) then
11             T_1 ... T_d ← split (T_a, T_b)
12             for i = 1...d do
13                 Q.put (T_i)
14             remove T_b from T(a,t)
15             if select (T_a, z) then
16                 send "detect e T_a" to z
17         else
18             insert T_a in T(a,t)
```

hypersphere-test fails, the $d-$simplex T_a is stored (line 18). In the opposite case, T_a is split and the recursive process is achieved by putting the resulting $d-$simplices in the queue (lines 12-13). In this case, a `detect` message is sent to the new entity z (lines 16) if the entity is the selected one (line 15).

3.3 Analysis

We begin the analysis of the algorithm by the computation cost. Indeed, the number of in-hypersphere-test to perform is a main concern as it is an expensive task.

We note k the number of neighbors of the new entity. Each neighbor of the new entity, except the $d+1$ entities in the enclosing $d-$simplex, has been detected by its d neighbors. Each detection requires one in-hypersphere-test, so $d*(k-(d+1))$ operations should be performed. Moreover, the insertion requires d additional failing in-hypersphere-tests by entity before to end the algorithm. So, $k*d$ additional in-hypersphere-tests are necessary. Therefore, the total number of in-hypersphere-tests is less than $2*d*k$.

We then show that the computation task is fairly distributed among the neighbors. The worst case

is as follows: one entity p_0 is linked with all of the neighbors of z, so one entity participates to all the $d-$simplices to discover. In this case, this entity should realize $k-(d+1)$ times the in-hypersphere-test as the $d+1$ entities generating the enclosing $d-$simplex do not require any test. But, d additional in-hypersphere-tests end the algorithm. Therefore, the number of in-hypersphere-tests performed by one neighbor of a new entity linked to k neighbors is less than k.

We now deal with the time complexity. We consider asynchronous communication links, so it is impossible to give a bound on the time required for the insertion of a new entity. However, it is possible to measure the number of *causal operations*: the number of times the new entity should send a `hello` after reception of a `detect` until it discovers all neighbors. We show that the maximal number of causal operations required by the insertion of a new entity linked to k neighbors is less than $\frac{k-d}{2}$.

Let z be the new entity and $\{p_0, p_1, \dots p_i, \dots p_k\}$ the set of neighbors of z in $DT(t+1)$. The worst case occurs when the length of the longest path between z and the farthest entity is maximal in $DT(t)$. That is, this situation occurs when the enclosing $d-$simplex is $(p_0, p_1, \dots p_{d-2}, p_k)$. In this case, the farthest entity is the entity which lies on the middle of the path between p_{d-2} and p_k, so the entity $p_{\frac{k-(d-2)}{2}}$. The d first entities are discovered without any causal operations. On the contrary, all following detected neighbor requires one causal operation, so at worst $\frac{k-(d-2)}{2}$ operations.

4 Entity Deletion

We now focus on the deletion of an entity. Entities which leave the system can quickly compute the new connections and inform their neighbors about the new links they have to create. Thus, when the set of neighbors of the faulty entity is known, the optimal computation of the new Delaunay triangulation has a complexity of $\mathcal{O}(k \cdot log\ k)$ where k is the number of neighbors of the entity [8]. Other robust algorithms admit a complexity of $\mathcal{O}(k^2)$ but simpler implementations [17].

However, entities may crash with no graceful behavior. The crash of an entity $z \in V(t)$ at time t is noticed either by a failure detector, either by the reception of a message from a neighbor. This crash generates a *hole* in the triangulation but no existing connection can be altered by the failure.

First we describe a naive approach, then we present an improved algorithm, which reduces the number of messages sent and the computation cost, especially by reducing the number of *in-hypersphere-tests*.

4.1 Basic Generalized Algorithm

A very basic approach to fill the hole emerged from the crash of z would be to let each neighbor of z collect all entities on the border of the hole, then recompute the triangulation. However, a neighbor of z in $DT(t)$ may not know all of its neighbors in $DT(t+1)$. So, two new messages, `neighbor` and `onborder` are introduced. They aim to provide to each neighbor of z the complete set $K(z,t)$.

We consider an entity $a \in K(z,t)$ detecting the crash of the entity z. It immediately sends a message `neighbor` to its neighbors sharing with itself a simplex with z. Let $T_z(a,t)$ be the set of simplices in $\mathcal{T}(a,t)$ which contain z. The message `neighbor` is sent to all entities in $K(a,t)$ occurring in at least one simplex in $T_z(a,t)$ This message contains a description of itself and the identifier of z.

We consider now an entity $b \in K(z,t) \cap K(a,t)$ receiving the message `hello` sent by a. This message is forwarded to the neighbors of b sharing a simplex with z, so to all entities in $K(b,t)$ occurring in a simplex in $T_z(b,t)$.

Assume now an entity c not linked to a but receiving the forwarded message `neighbor`. The entities a and c are both neighbors of the faulty entity z in $DT(t)$. So, they have to know each other. The entity c sends a message `onborder` to a. This message contains all entities in $K(c,t)$ occurring in $T_z(c,t)$. Thus, the entity a knows all entities involved in a simplex containing z. Therefore, the entity a can easily determine if it still has to wait for some new messages `onborder`.

With this flooding mechanism, the entity a even-

tually knows all entities in $K(z,t)$. So, it can use any triangulation algorithm to reconstruct the triangulation and contact its new neighbor.

This process has to be executed by all neighbors of z although the triangulation computation is expensive. Moreover, the flooding generates a lot of messages. In the next section we present an improved algorithm which does not require the full knowledge of $K(z,t)$. Moreover, this algorithm takes advantage of the $(d-1)$-simplices forming the boundary of the hole in order to update the triangulation.

4.2 Improved Generalized Algorithm

We first describe the outlines of the algorithm. Then, we present it in in the simplest case: the faulty entity z has only $d+1$ neighbors. Finally, we detail a non-trivial example in a three-dimensional space.

4.2.1 Description

Assume an entity $a \in K(z,t)$ detecting the crash of z. A description of the treatment is given in Algorithm 2. Its first task (lines 1-7) consists of extracting from $\mathcal{T}(a,t)$ a set $T_z(a,t)$ of d-simplices containing z. By removing z from them, a knows some $(d-1)$-simplices which belongs to the boundary of the *hole* generated by the crash of z. We note $T_z^\star(a,t)$ the set of these simplices.

Then, the entity a builds a *candidate* d-simplex using each pair of $(d-1)$-simplices sharing a common $(d-2)$-simplex (lines 8-11). The candidates are communicated with the message `candidate` to all entities in $K(a,t)$ involved in at least one simplex in $T_z(a,t)$ (lines 12-13).

We consider now an entity $b \in K(z,t)$ receiving a message `candidate` from the entity a. It first performs a in-hypersphere-test with the candidate $d-$simplex T if it is not directly involved in T. If the test fails — if b is not within $\mathcal{C}(T)$ — b forwards the message to the entities in $K(b,t)$ which belong to at least one simplex in $T_z(b,t)$ and do not participate to the candidate $d-$simplex T. This way,

Algorithm 2: a detects the crash of z

```
1  T_z ← {T ∈ T(a,t) : z ∈ T}
2  K_z ← {e ∈ T : T ∈ T_z}
3  T_z* ← ∅
4  foreach T ∈ T_z do
5      remove T from T(a,t)
6      T* ← T \ {z}
7      insert T* in T_z*
8  foreach (T_i, T_j) ∈ T_z*, i ≠ j do
9      if |T_1 ∩ T_2| = d − 2 then
10         new_T ← (T_1 ∪ T_2)
11         insert new_T in T(a,t)
12         foreach e ∈ K_z do
13             send "candidate new_T a" to e
```

the message `candidate` turns around the hole and no entity within the boundary may miss this new simplex.

Algorithm 3: `candidate-fail` T c

```
1  K_z ← K_z ∪ c
2  insert c in K(a,t)
3  T_1, T_2 ← (T_t ∈ T_z* : T_t ⊂ T)
4  foreach T_i ∈ {T_1, T_2} do
5      new_T ← T_i ∪ {c}
6      foreach e ∈ K_z do
7          send "candidate new_T a" to e
```

If the in-hypersphere-test succeeds — $b \in \mathcal{C}(T)$ — the entity b sends a message `candidate-fail` to a. This message contains the failed simplex and a description of the emitter. Upon reception of a message `candidate-fail` (see Algorithm 3), the entity a first retrieves the $(d-1)$-simplices used to build the failed candidate (line 3). Then, it builds a new candidate d-simplex from these $(d-1)$-simplices and the entity c (lines 4-5). Finally, it communicates these new candidates with a message `candidate` to its neighbors on the hole boundary (lines 6-7).

4.2.2 A Trivial Case

The *hole* left by z is exactly the missing d-simplex T in $DT(t+1)$. No further actions are needed ex-

cept that all neighbors of z should add T into their respective buffer $\mathcal{T}$. Indeed, the $d+1$ neighbors of z are already interconnected.

The entity z participates to $d+1$ d-simplices. Each neighbor of z is involved in d out of these $d+1$ d-simplices. The d d-simplices known by $e_i \in K(z,t)$ are noted :

$$T_0^{e_i} = \{e_1, \ldots, e_d, z\}$$
$$\vdots$$
$$T_j^{e_i} = \{e_k | 0 \leq k \leq d\} \cup \{z\} \setminus \{e_j\} \quad j \neq i$$
$$\vdots$$
$$T_d^{e_i} = \{e_0, \ldots, e_{d-1}, z\}$$

Note that $T_i^{e_i}$ does exist, but is not in $\mathcal{T}(e_i, t)$.

For the entity e_i, the hole is circumscribed by the d $(d-1)$-simplices resulting from the operation $T^{e_i} \setminus \{z\}$. Only one of the $d+1$ $(d-1)$-simplices ($T_i^{e_i}$) is not known by e_i. Yet T can be easily rebuilt by any pair of the $(d-1)$-simplices:

$$T = \{T_m \cup T_n\} \quad (0 \leq m, n \leq d,\ m, n \neq i)$$

For better understanding, let us rephrase this basic idea in the 3-dimensional space: the hole has the shape of a tetrahedron. It is circumscribed by four triangles. The four entities can be extracted from any pair of triangles. So, the missing tetrahedron can be built from any pair of triangles.

The resulting d-simplex T is identical for each pair of the $(d-1)$-simplices. Actually the result is only a *candidate* simplex. Indeed, it can not be in conflict with any entity in $K(e_i, t)$, but it may be with an entity in $K(z, t)$. So, the entity e_i sends to the d neighbors $K(z,t) \cap K(e_i, t)$ a message *candidate* containing T.

Upon reception of a message `candidate`, the entities do not have to execute the in-hypersphere-test because they participate to the candidate d-simplex T. They have to forward the message to their neighbors that are both neighbors of z and not included in T. In this trivial case, there is no entity to test, as all neighbors involve in the candidate. So, the recursion immediately stops. The candidate simplex T is validated but no new connections are required.

4.2.3 Non-Trivial Example

The following example relies on the situation depicted in Figure 6 and Figure 7. The entities a, b, d and f lie in front, while the entities c and e are in the back. The faulty entity z is inside the convex hull of all these entities. The entity z is involved in 8 d-simplices (tetrahedra): (a,b,d,z), (a,b,e,z), (a,e,f,z), (a,d,f,z), (b,c,d,z), (b,c,e,z), (c,d,e,z), and (d,e,f,z). The $(d-1)$-simplices (triangles) defining the hole left by z are: (a,b,d), (a,b,e), (a,e,f), (a,d,f), (b,c,d), (b,c,e), (c,d,e), and (d,e,f).

Consider the entity a for the rebuilding process: a is involved in 4 of the 8 tetrahedra. Consequently it only knows 4 of the 8 triangles surrounding the hole. Its candidate simplices are $T_1^a = (a,b,d,e)$, $T_2^a = (a,b,d,f)$, $T_3^a = (a,b,e,f)$, and $T_4^a = (a,d,e,f)$. They are constructed from each pair of the triangles sharing one edge. Then, a sends some messages `candidate` containing T_i^a $(i = 1,2,3,4)$ to its neighbors in $K_z(a,t)$, so b, c, d, e, and f.

We now focus on the message `candidate` containing T_1^a received by b. The entity b participates to this candidate, so it just forwards the message to c because c is the only neighbor of b belonging to the boundary of the hole and not involved in T_1^a.

The entity c does not participate to the candidate simplex T_1^a. Therefore, it should perform a in-hypersphere-test. If c is not in $\mathcal{C}(T_1^a)$, nothing happens and the recursion stops. On the contrary, the entity c should send a message `candidate-fail` message to a.

The candidate T_1^a was built using the triangles (a,b,d) and (a,b,e). As they do not form a Delaunay tetrahedra together, the entity a uses it to build two new candidate simplices with the new neighbor c: $T_{11}^a = (a,b,d,c)$ and $T_{12}^a = (a,b,e,c)$. Then, it communicates these new candidates to its neighbors in the boundary of the hole, including its new neighbor c.

With the failed candidate T_4^a a can build the new candidates $T_{41}^a = (a,c,d,f)$ and $T_{42}^a = (a,c,e,f)$.

The final Delaunay triangulation after the deletion of z contains five tetrahedra: (a,b,c,d), (a,b,c,e), (a,c,d,f), (a,c,e,f), and (c,d,e,f).

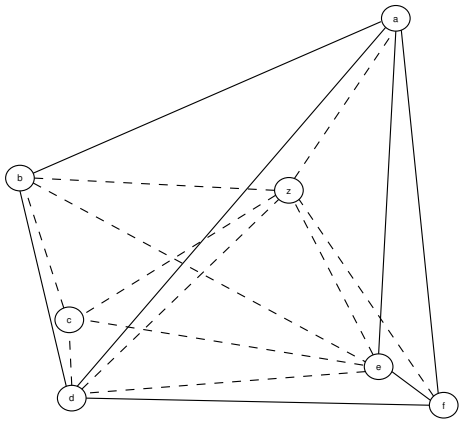

Figure 6: Before the deletion of z.

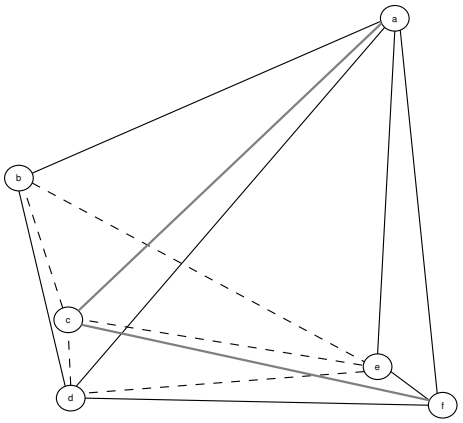

Figure 7: After the deletion of z.

Two new connections are established: (a, c) and (f, c) as shown in Figure 7.

4.3 Analysis

In comparison with the basic algorithm, the improved generalized algorithm does not require that each neighbor of the faulty entity knows all neighbors of z. Moreover, it does not require that each neighbor of z computes an updated triangulation.

The algorithm fills the hole from its boundary to its center. At first, candidate simplices are built from existing $(d-1)$-simplices. Then, the reception of the messages `candidate-fail` forces entities to compute some new simplices and to open new connections. In the worst case, the iterative process halts when each entity knows all neighbors of the crashed entity. So, the improved algorithm is, at worst, so expensive than the basic one and, in the vast majority of case, more efficient.

5 Conclusion

In this paper, we present a set of algorithms for the dynamic maintaining of a distributed overlay network matching with the Delaunay triangulation of the entities. The entities can have a position in any $d-$dimensional space. We consider here the arrival of new entities and the crash of an entity.

We originally aim to apply the protocol in $3-$dimensions for shared virtual worlds. Especially, we intend to use the Solipsis [13] platform in order to implement the protocol. Current version of the Solipsis protocol is light, however it does not guarantee a perfect coherency in the virtual world. A Delaunay-based overlay could guarantee it, but the computation cost could become an issue if entities are very dynamic. Among the future works, we intend to evaluate the protocol and to compare it with the Solipsis protocol in a context of real humans controlling virtual entities.

One of the main drawbacks of these algorithms is the greedy walk for the detection of the closest entity to the queried position. Some recent studies aim to construct small-world networks by adding only one edge between two entities in the overlay. In these small-world networks, a basic walk is guaranteed to succeed in polylogarithmic time. In future works, we will try to transform the Delaunay triangulation to a small-world network, such that the detection of the closest entity could be substantially more efficient.

Acknowledgment

We would like to thank Antoine Pitrou who has the first hints of the deletion algorithm and Joaquín Keller who motivates this study.

References

[1] A.Bharambe, S. Rao, and S. Seshan. Mercury: a Scalable Publish-Subscribe System for Inter-

net Games. In *Workshop on Network and System Support for Games (Netgames'02)*, 2002.

[2] F. Araujo and L. Rodrigues. Geopeer: A location-aware peer-to-peer system. Technical report, Faculdade de Ciências da Universidade de Lisboa, Portugal, 2001.

[3] F. Aurenhammer and R. Klein. Voronoi Diagrams. In *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishers, 2000.

[4] J. D. Boissonnat and M. Teillaud. The Hierarchical Representation of Objects: the Delaunay Tree. In *ACM Symp. on Computational Geometry*, 1986.

[5] P. Bose and P. Morin. Online Routing in Triangulations. In *International Symposium on Agorithms and Computation*, 1999.

[6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a Decentralized Network Coordinate System. In *ACM SIGCOMM*, 2004.

[7] B. Delaunay. Sur la sphère vide. *Otdelenie Matematicheskii i Estetvennyka Nauk*, 7:793–800, 1934.

[8] O. Devillers. On Deletion in Delaunay Triangulations. In *Symp. on Computational Geometry*, 1999.

[9] H. Edelsbrunner and N. R. Shah. Incremental Topological Flipping Works for Regular Triangulations. *Algorithmica*, 15:223–241, 1996.

[10] P. Green and R. Sibson. Computing Dirichlet Tesselations in the Plane. *The Computer Journal*, 21:168–173, 1978.

[11] L. Guibas, D. Knuth, and M. Sharir. Randomized Incremental Constructions of Delaunay and Voronoi Diagrams. *Algorithmica*, 7:381–413, 1992.

[12] S-Y. Hu and G-M. Liao. Scalable Peer-to-Peer Networked Virtual Environment. In *Network and Systems Support for Games (NetGames'04)*, 2004.

[13] J. Keller and G. Simon. Solipsis: A Massively Multi-Participant Virtual World. In *Int. Conf. on Parallel and Distributed Techniques and Applications (PDPTA'03)*, 2003.

[14] I. Lee and V. Estivill-Castro. Polygonization of Point Clusters through Cluster Boundary Extraction for Geographical Data Mining. In *Int. Symposium on Geospatial Theory, Processing and Applications*, 2002.

[15] X.Y. Li, G. Calinescu, and P-J. Wan. Distributed Construction of a Planar Spanenr and Routing for Ad Hoc Wireless Networks. In *Proceedings of IEEE Infocom*, 2002.

[16] J. Liebeherr and M. Nahas. Application-layer multicasting with delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications*, 20(8):1472–1488, October 2002.

[17] M-A. Mostafavia, C. Goldb, and M. Dakowiczb. Delete and Insert Operations in Voronoi/Delaunay: Methods and Applications. *Computers & Geosciences*, 29:523–530, 2003.

[18] T. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *INFOCOM'02*, 2002.

[19] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons, 2000.

[20] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Virtual Landmarks for the Internet. In *International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[21] M. I. Shamos and D Hoey. Closest-Point Problems. In *IEEE Symposium on Found. Comput. Sci. (FOCS'75)*, pages 151–162, 1975.

[22] M. Szymaniak, G. Pierre, and M. van Steen. Latency-Driven Replica Placement. In *Int. Symposium on Applications and the Internet (SAINT'05)*, 2005.

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

[24] D. F. Watson. Computing the n-Dimensional Delaunay Triangulation with Application to Voronoi Polotypes. *The Computer Journal*, 24(2):167–172, 1981.

[25] Z. Xu, C. Tang, S. Banerjee, and S.-J. Lee. Receiver Initiated Just-In-Time Tree Adaptation for Rich Media Distribution. In *NOSSDAV*, 2003.