Looking for a Definition of Dynamic Distributed Systems*

R. Baldoni[1], M. Bertier[2], M. Raynal[2], and S. Tucci-Piergiovanni[1]

[1] IRISA, Campus de Beaulieu, 35042 Rennes, France
[2] Computer Science Department, University *La Sapienza*, Roma, Italy
{marin.bertier,raynal}@irisa.fr,
{baldoni,sara.tucci}@dis.uniroma1.it

Abstract. This paper is a position paper on the nature of dynamic systems. While there is an agreement on the definition of what a static distributed system is, there is no agreed definition on what a dynamic distributed system is. This paper is a first step in that direction. To that end, it emphasizes two orthogonal dimensions that are present in any dynamic distributed system, namely the varying and possibly very large number of entities that currently define the system, and the fact that each of these entities knows only a few other entities (its neighbors) and possibly will never be able to know the whole system it is a member of. To illustrate the kind of issues one has to cope with in dynamic systems, the paper considers, as a "canonical" problem, a simple data aggregation problem. It shows the type of dynamic systems in which that problem can be solved and the ones in which it cannot be solved. The aim of the paper is to give the reader an idea of the subtleties and difficulties encountered when one wants to understand the nature of dynamic distributed systems.

1 Introduction

The nature of distributed computing. Distributed computing arises when the problem to solve involves several entities such that each entity has only a partial knowledge of the many parameters involved in the problem. According to the context, these entities are usually called processes, nodes, sites, sensors, actors, peers, agents, etc. The entities communicate and exchange data through a communication medium (usually an underlying network).

While *parallelism* and *real-time* can be respectively characterized by the words "efficiency" and "on time computing", distributed computing can be characterized by the word "uncertainty". This uncertainty is created by asynchrony, failures, unstable behaviors, non-monotonicity, system dynamism, mobility, low computing capability, scalability requirements, etc. Mastering one form or another of uncertainty is pervasive in all distributed computing problems. So, a fundamental issue of distributed computing consists in finding concepts and mechanisms that are general and powerful enough to allow reducing (or even eliminating) the underlying uncertainty.

* This work has been done in the context of the European Network of Excellence ReSIST (Resilience for Survivability in IST).

V. Malyshkin (Ed.): PaCT 2007, LNCS 4671, pp. 1–14, 2007.

Static reliable asynchronous distributed systems. A distributed system (the software and hardware layer on top of which the distributed applications are executed) can be characterized by behavioral properties and structural properties. These properties define a computation model.

The static reliable asynchronous model is the most popular one. *Static* means that the number of entities is fixed. *Reliable* means that neither the entities nor the communication medium suffer failures. *Asynchronous* means that there is no particular assumption on the speed of the processes, or on message transfer delays. Moreover, the underlying network is usually considered as fully connected: any entity can send messages to, or receive messages from, any other entity (this means that the message routing is hidden at the abstraction level offered by this distributed computing model).

An important result associated with this distributed computing model is the determination of a consistent global state (sometimes called a snapshot). It has been shown [5] that the "best" that can be done is the computation of a global state (of the upper layer distributed application) with the following consistency guarantees: the computed global state is such that (1) the application could have passed through it, but (2) has not necessarily passed through it. There is no way to know whether or not the actual execution passed through that global state. This is one of the fundamental facets of the uncertainty encountered in static distributed systems.

Static unreliable asynchronous distributed systems. The simplest static unreliable asynchronous model is characterized by the fact that processes may crash. The most famous result for this model is the impossibility to solve the consensus problem as soon as a process may crash [6] (the consensus problem is a coordination - -or agreement- - problem. It consists in designing a deterministic protocol in which all the processes that do not crash reach a common decision based on their initial opinions). The impossibility to solve this problem comes from the net effect of asynchrony and failures. One way to solve consensus despite asynchrony and failures consists in enriching the asynchronous model with appropriate devices called failure detectors [3,10] (so, the resulting computing model is no longer fully asynchronous).

Fortunately, problems simpler than consensus can be solved in this model. Let us consider the reliable broadcast problem [8] as an example. This problem consists in providing the processes with a broadcast primitive such that all the processes that do not crash deliver all the messages that are broadcast (while the faulty processes are allowed to deliver only a subset of these messages). Let a correct process be a process that never crash. This problem can easily be solved as soon as any two correct processes remain forever connected through a path made up of reliable channels and correct processes.

So, when we proceed from the static reliable asynchronous distributed computing model to its unreliable counterpart, there are problems that can still be solved, while other problems become impossible to solve if asynchrony is not restricted (e.g., by using failure detectors, or considering their "ultimate" endpoint, namely, a synchronous system).

Dynamic distributed systems. Since a recent past, there are a lot of papers (mainly in the peer-to-peer literature) that propose protocols for what they call *dynamic systems*. These protocols share the following: the entities can join and leave the system at will. This

dynamicity dimension constitutes a new attribute of the uncertainty that characterizes distributed computing. Unfortunately, (to our knowledge) there is no clear definition of what a dynamic system is. This paper is a first step in that direction. To that end, it proposes to investigate two dimensions of dynamicity. The first is on the number of entities that compose the system: is there an upper bound that is known? How many entities can coexist at any given time? etc. The second dimension is "geographical". More precisely, it is related to the fact that it is not possible to provide the entities with an abstraction offering a logical point-to-point bidirectional link to each pair of entities. So, this dimension is on the notion of entity neighborhood (locality) and the fact that the processes can or cannot know an upper bound on the network diameter.

Content of the paper. The paper is made up of 4 sections. Section 2 proposes parameters that should be taken into account when one wants to precisely define a dynamic system model. Considering a very simple dynamic system, Section 3 investigates what can be computed in this model. To that end a simple aggregation problem is used as a "canonical" problem. Section 4 provides a few concluding remarks.

The spirit of the paper is more the spirit of a position paper with a pedagogical flavor than the spirit of a traditional research paper. We do think that a precise definition of what a dynamic distributed system is (or maybe what families of dynamic distributed systems are) is hardly needed. This paper is a very first endeavor towards this goal.

2 Elements for Defining a Dynamic Distributed System

Informally, *a dynamic system is a continually running system in which an arbitrarily large number of processes are part of the system during each interval of time* and*, at any time, any process can directly interact with only an arbitrary small part of the system.* This section proposes and investigates two attributes that should be part of the definition of any dynamic distributed system.

2.1 Modeling the Dynamic Size of the System in Terms of Number of Entities

In a dynamic system, entities may join and leave the system at will. Consequently, at any point on time, the system is composed of all processes (entities) that have joined and have not yet left the system. We call *system run* (or simply a run) a total order on the join and leave events (issued by the processes) that respect their real time occurrence order.

In order to model entities continuously arriving to and departing from the system, we assume the infinite arrival model (as defined in [9]), where, in each run, infinitely many processes $P = \{\ldots, p_i, p_j, p_k \ldots\}$ may join the system. However, several models can be defined, that differ in the assumptions on the number of processes that can *concurrently* be part of the system [7,9]. Using the notation introduced in [1], the following infinite arrival models can be defined:

- M^b: The number of processes concurrently inside the system is bounded by a constant b in all runs.
- M^n: The number of processes concurrently inside the system is bounded in each run, but may be unbounded when we consider the union of all the runs.

– M: The number of processes that join the system in a single run may grow to infinity as the time passes.

In the first model, the maximum number of processes in each run is bounded by a constant b that is the same for all the runs. When it is known, that constant can be used by the protocols defined for that system.

In the second model, the maximum number of processes in each run is bounded, but that bound may vary from one run to another. It follows that no protocol can rely on such a bound as a protocol does not know in advance the particular run that will be produced.

In the third model, the number of processes concurrently inside the system is finite when we consider any finite time interval, but may be infinite in an infinite interval of time. This means that the only way for a system to have an infinite number of processes is the passage of time.

2.2 Modeling the Dynamic Size of the System in Terms of Geography

The previous models [7,9] implicitly assume that, at any time, the communication network is fully connected: any process knows any other process that is concurrently in the system, and can send it - -or receive from it- - messages directly through a point-to-point channel.

Our aim is here to relax this (sometimes unrealistic) assumption, and take into account the fact that, at any time, each process has only a partial view of the system, i.e., it can directly interact with only a subset of the processes that are present in the system (this part is called its *neighborhood*). So, we consider the following *geographical* attributes for the definition of a dynamic distributed system.

– At any time, the system can be represented by a graph $G = (P, E)$, where P is the set of processes currently in the system and E is a set of pairs (p_i, p_j) that describe a symmetric neighborhood relation connecting some pairs of processes. $(p_i, p_j) \in E$ means that there is a bidirectional reliable channel connecting p_i and p_j.
– The dynamicity of the system, i.e., the arrivals and departures of processes, is modeled through additions and removals of vertices and edges in the graph.
  • The addition of a process p_i to a graph G brings to another graph G' obtained from G by including p_i and a certain number of new edges (p_i, p_j) where the p_j are the processes to which p_i is directly connected.
  • The removal of a process p_i to a graph G brings to another graph G' obtained from G by suppressing the vertex p_i and all the edges involving p_i.
  • Some new edges can be added to the graph, and existing edges can be suppressed from the graph. Each such addition/deletion brings the graph G into another graph G'.
– Let $\{G_n\}_{run}$ denote the sequence of graphs through which the system passes during a given run. Each $G_n \in \{G_n\}_{run}$ is a *connected graph the diameter of which can be greater than one* for all runs.

As we have seen, an infinite arrival model allows capturing a dynamicity dimension of dynamic distributed systems. Making different assumptions on the diameters of the

graphs in the sequences $\{G_n\}_{run}$ allows capturing another dynamicity dimension related to the "geography" of the system. More specifically, we consider the following possible attributes. In the following $\{D_n\}_{run}$ denotes the set of the diameters of the graphs $\{G_n\}_{run}$.

- *Bounded and known diameter*. In this case the diameter is always bounded by b, i.e., for each $D_n \in \{D_n\}_{run}$ we have $D_n \leq b$ for all the runs, and that bound is known by the protocols designed for that model.
- *Bounded and unknown diameter*. In this case all the diameters $\{D_n\}_{run}$ are finite in each run, but the union of $\{D_n\}_{run}$ for all runs can be unbounded. In that case, as an algorithm cannot know in which run it is working, it follows that the maximal diameter remains unknown to the protocol. So, in that model, a protocol has no information on the diameter.
- *Unbounded diameter*. In this case, the diameter is possibly growing indefinitely in a run, i.e., the limit of $\{D_n\}_{run}$ can go to infinity.

2.3 Dynamic Models Definition

A model is denoted as $M^{N,D}$ where N is on the number of processes and D is on the graph diameter, both parameters can assume the value b, n, ∞ to indicate respectively a number of entities/diameter never exceeding a known bound, a number of entities/diameter never exceeding an unknown bound and a number of entities/diameter possibly growing indefinitely (in the following, if a parameter may indifferently assume any value, we denote that as $*$). Possible models are $M^{b,b}$, $M^{n,b}$, $M^{\infty,b}$ ([1]), $M^{n,n}$, $M^{\infty,n}$ and $M^{\infty,\infty}$.

Note that the previous models characterize only the *dynamicity* of the system without considering other more classical aspects such as the level of synchrony or the type of failures. Clearly, any of these models can be refined further by specifying these additional model attributes as usually done in static systems.

To be able to establish the impact of geographical assumptions on a problem solving in dynamic distributed systems, we only consider, in this paper, synchronous systems or asynchronous system completed with perfect failure detectors. In other words, we assume that a node can have reliable information about nodes in its neighborhood.

3 An illustrating Example: One-Time Query

3.1 The One-Time Query Problem

To illustrate and investigate the previous attributes of a dynamic distributed system, we consider the *One-Time Query* problem as defined in [2]. This problem can informally be defined as follows. A process (node) issues a query in order to aggregate data that are distributed among a set of processes (nodes). The issuing process does not know (i) if there exist nodes holding a value matched by the query, (ii) where these nodes are, (iii) how many they are. However, the query has to complete in a meaningful way in spite of the uncertainty in which the querying node works.

[1] An instance of the model $M^{\infty,b}$ is M^{∞} of [1] where the diameter is implicitly set to 1.

The One-Time Query problem, as stated in [2] requires that the query, issued by a node p_i aggregates *at least* all the values held by the nodes that are in the system and are connected to p_i during the whole duration of the query (query time interval).

Unfortunately, this specification has been intended for a model slightly different from the more general model proposed in the previous section. In fact, the system is intended to be *monotonous* in the sense that it can be represented by a graph G defined at the beginning of the computation (query) and from which edges can be removed as time passes, but to which no new edges can be added as time passes. Differently, in the previous models, the system is *dynamic* in the sense that nodes/edges additions and nodes/edges deletions are allowed. As we are about to see, while the One-Time Query problem -as defined above- cannot be solved in a dynamic system, a weaker version of it can be. It is also important to notice (as we will show later) that this weaker version cannot be solved in $M^{\infty,\infty}$.

One-Time Query specification. The specification that follows is due to [2]. Let $query(Q)$ denote the operation a process invoke to aggregate the set of values $V = \{v_1, v_2, \ldots\}$ present in the system and that match the query. The aim for the process that issues the query is to compute $v = Q(V)$. Given that setting, the problem is defined by the following properties (this means that any protocol solving the problem has to satisfy these properties):

- Termination: $query(Q)$ completes in a finite time.
- Validity: The set V of data obtained for computing $query(Q)$ includes at least the values held by processes that are member of the system during the whole query time interval.

3.2 The WILDFIRE Algorithm

In [2] the following algorithm (called WILDFIRE) to solve the problem is proposed. This algorithm relies on the following assumptions:

- synchronous channels with a known upper bound δ,
- a known upper bound on the network diameter D.

Algorithm description. The principle of this algorithm is simple. Each process which receives a so-called query-update message updates its current value to a new one, computed by aggregating the current value and the received value, then it spreads the new value to its neighbors.

The initiator of the query just sends its initial value to its neighbors in a query-update message and waits for at least $2 * D * \delta$ time before returning its value. $D * \delta$ is the time required to inform all nodes in the network about the query, and the same duration is required to transmit values to the initiator.

As the initiator, all nodes which receive a query-update message for the first time, initiate a timeout and when this timeout expires, they stop to process all new query-update messages.

In [2], the authors propose to reduce the number of messages exchanged by sending a query-update message only when there is new information: (i) if the remote value

doesn't change the local value, then the node doesn't send any message (except for the first reception of the query-update message), (ii) if the aggregate value is equal to the remote one, then the node transmits the new value to its neighbors except the sender of the remote value.

```
INITIALIZATION
1  active ← false;
2  v ← initial_value;

LAUNCH(Q)
3  active ← true;
4  d ← D; % D is the upper bound on the network diameter. %
5  send [QUERY-UPDATE (Q, d − 1, v)] to neighbors;
6  set timeout T ← 2d ∗ δ;
7  when (T elapses) do
8    active ← false;
9    return (v);

RECEPTION
10 when (receive [QUERY-UPDATE(Q, d, rv)] from p_j ) do
11   if (¬active)
12     then set timeout T ← 2d ∗ δ; % We consider negligible process step's executions w.r.t. message delays. %
13   if (T not yet elapsed)
14     then temp ← aggregate(v, rv);
15          if (temp! = v or ¬active)
16            then active ← true; v ← temp;
17                 send [QUERY-UPDATE,(Q, d − 1, v)] to neighbors − p_j ;
18                 if (v! = rv)
19                   then send [QUERY-UPDATE,(Q, d − 1, v)] to p_j
```

Fig. 1. The WILDFIRE Algorithm

3.3 The One-Time Query Problem for Dynamic Models

One-Time Query problem solvability. The WILDFIRE algorithm solves the one-time query problem in a monotonous network but does not solve it in a dynamic network (in none of the models presented in the previous section, neither in $M^{*,b}$). More generally, the one-time query specification introduced so far is too strong, and cannot be satisfied by any algorithm if the network graph can change by *adding* edges *during the query completion* [2]. However, if an edge is added during a query, the following bad scenario can happen.

Description on a bad scenario. Let us consider the querying process p_A and a process p_E (i) inside the system when the query starts and (ii) connected to p_A through a given path. Let us suppose that an edge joining p_A and p_E is *added after* the query started and remains up until the query ends. Let us also suppose that the path previously connecting p_A and p_E is removed (due to a crash of some process in the path) before the query ends. Formally, p_A is always connected to p_B throughout the entire duration of the query (as herein assumed by all dynamic models), but its value could not be retrieved as described in Figure 2 where t_q is the time the query starts.

[2] The addition of edges during the query completion is reasonable as the query takes an arbitrary long time spanning the entire graph and in order to maintain connectivity edges addition may be needed in spite of edges removals occurring at arbitrary times.

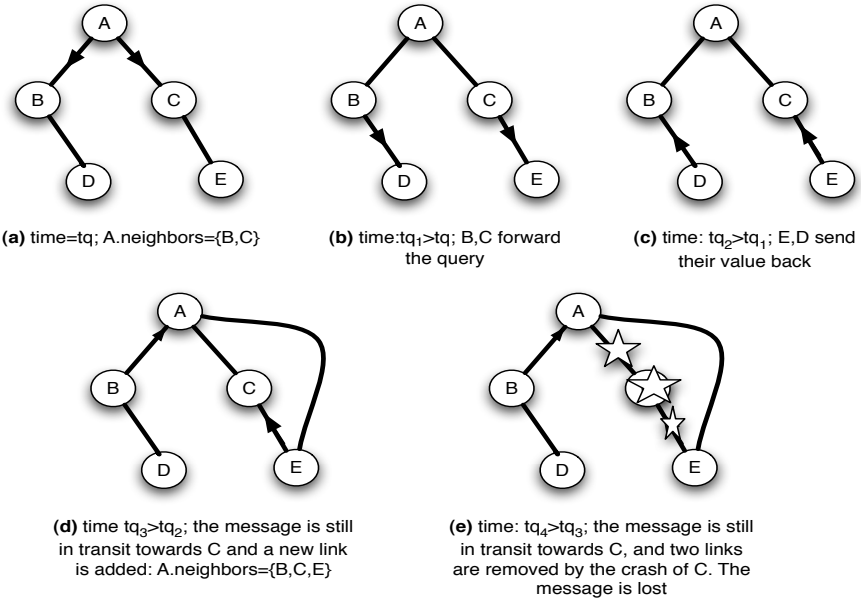

Fig. 2. Bug Example

The problem lies in the fact that the process p_E replies to the query but the message containing the reply is exchanged through a path that is removed before the query completes, and is consequently lost before it reaches the querying process.

Then, to retrieve this value, p_E should be forced to send again the reply back (this can be done by assuming a detection of the path removal that triggers a new sending on the new path). However, by the nature of the infinite arrival model, the substitution of a path with a new one during the query could happen infinitely often in all dynamic models in which the diameter is not bounded by one (see Fig. 3). In all these models the query may never complete violating termination.

One-Time Query specification for dynamic models. The specification of the one-time query problem in case of a dynamic model is here refined bringing to the definition of the Dynamic One-Time Query Specification. This new specification states that the values to include in the query computation are at least those coming from nodes that belong to the graph G defined at time the query starts, and remain connected, during the whole query interval, to the querying process through a subgraph of G. More formally, the Dynamic One-Time Query specification satisfies the following two properties:

- Termination: $query(Q)$ completes in a finite time.
- Dynamic Validity: For each run, $query(Q)$ will compute the result including in V at least the values held by each process that, during the whole query interval, remains connected to the querying process through a subgraph of the graph G that represents the network at the time the query is started.

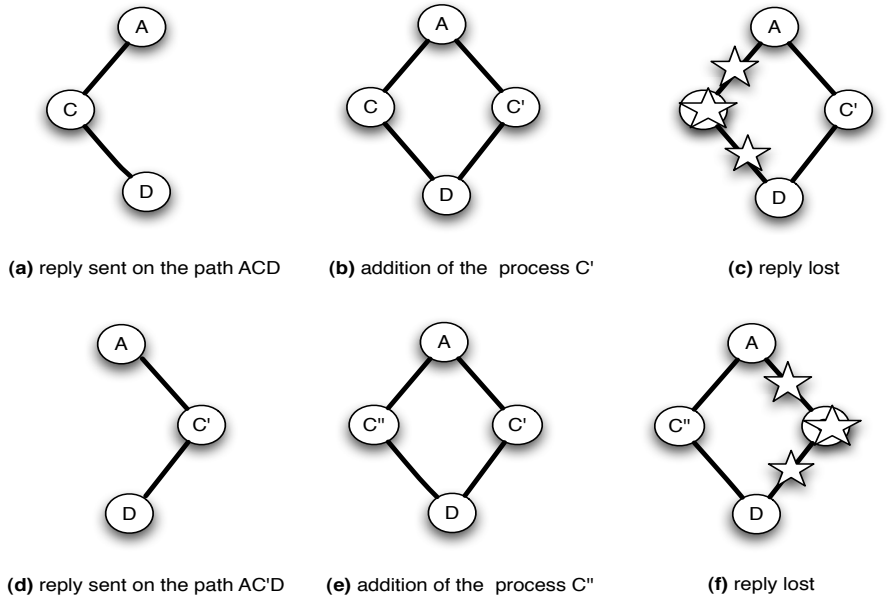

Fig. 3. Bad Pattern of Graphs Changing

It is important to note (and easy to see) that the dynamic one-time query specification is satisfied by the WILDFIRE algorithm in the model $M^{*,b}$ with $b > 1$. In the following we will explore if there exist solutions without assuming a known upper bound on the diameter.

3.4 The DEPTHSEARCH Algorithm

The algorithm that follows (called DEPTHSEARCH) solves the one-time query problem as defined just previously. That protocol relies on the following assumptions.

- asynchronous model enriched with a perfect failure detector (the faulty processes are deleted from the set $neighborhood$),
- unique process identifiers,
- a finite diameter of the network (not known in advance).

Algorithm description. This algorithm works in a different way than WILDFIRE. In WILDFIRE, many query-update messages are exchanged all over the network at the same time. In the DEPTHSEARCH algorithm only one message (query or reply) is transmitted at one time. The only case, in which two different queries co-exist, is the consequence of a disconnection between two nodes, but in any case only one query is taken into account.

This algorithm manages several sets:

- The set `values` that contains all values currently collected,
- The set `replied` that contains the identifiers of the nodes that have provided their value,

– `querying` contains the identifiers of the nodes that have sent a querying message and are waiting for replies from their neighborhood. These nodes (except the query initiator) are also nodes that have to provide their value to some other querying process.

This algorithm works similarly to a depth-first tree traversal algorithm (it traverses the nodes that compose the system). When a node p_i receives a query message, it checks if some of its neighbors have not yet received the query message yet by checking the querying and replied set. If some of them have not yet received a query message, then p_i sends to the first of them (say p_j) a query message and waits until it receives a reply from p_j.

When the node p_i receives a reply message from p_j, or if p_j is no more in the p_i's neighborhood (p_i is failed or is disconnected), the node p_i sends a query message to the next neighbor that has not yet received a query message. When all p_i's neighgbors have received a query message or are no longer in the p_i's neighborhood, then p_i sends back a reply message with the values and replied set updated or, if p_i is the query initiator, it returns the set of values.

```
INITIALIZATION
1  querying ← ∅;    % set of processes forwarding the query %;
2  replied ← ∅;    % set of processes replied to the query %;
3  targets ← ∅;    % set of processes to query by the local process %;
4  values ← {local_value};    % set of processes to query by the local process %;
5  neighborhood    % set of correct neighbors provided and updated by the perfect failure detector %

REQUEST(Q)
6  targets ← neighborhood; % This line freezes the neighbor set %;
7  querying ← querying ∪ {local_id};
8  for each i := 1 to |targets|
9     if (targets[i] ∉ {querying} ∪ {replied})
10       then send [QUERY,(Q, querying, replied)] to n[i];
11            wait until (receive [REPLY,r_values, r_replied] from n[i] ∨ n[i] ∉ neighborhood);
12            if (n[i] ∈ neighborhood)
13              then values ← values ∪ r_values;
14                   replied ← replied ∪ r_replied

LAUNCH(Q)
15 REQUEST(Q);
16 return (values)

RECEPTION
17 when (receive [QUERY,(Q, r_querying, r_replied)] from p_j ) do
18    querying ← r_querying;
19    replied ← r_replied;
20    REQUEST(Q);
21    replied ← replied ∪ {local_id};
22    send [REPLY,(values, replied)] to p_j;
```

Fig. 4. The DEPTHSEARCH Algorithm

Algorithm illustration. To illustrate the protocol behaviour, let us consider the computation related to a query initiated by a node p_A in the network shown in Figure 5. In this scenario p_A starts to query the first process in the $p_A.targets = (B, C, D)$ set, p_B does the same with its $p_B.targets = (A, C, E)$ set where $p_B.querying =$

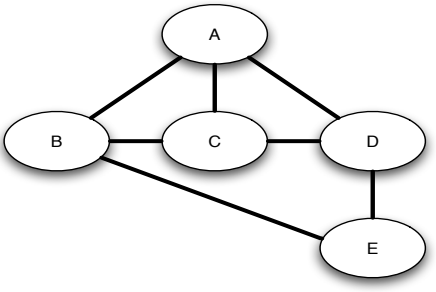

Fig. 5. Graph Representing the Network during the A's Query

$p_A.querying = \{A\}$. Then p_B queries p_C, piggybacking the list of querying processes, that now is $\{$A,B$\}$. Then, p_C does not query anyone and gives back a reply to p_B, with a list of $replied$ processes equal to $\{$C$\}$. At this point, p_B queries p_E. Let us consider the case in which the edge (p_B, p_E) breaks, then p_B ends and becomes part of $replied$ giving back to p_A the value come from p_B, v_b, and the value from p_C, v_c. Then, p_A avoids to query p_C as part of $replied$ and queries directly p_D, piggybacking the list of $p_A.querying$ still containing only A and the list of $replied$ equal to $\{$B,C$\}$. Then p_D avoids to query p_C and it queries only p_E. p_E receives the query with the following information: querying processes $\{$A,D$\}$, replied processes $\{$B,C$\}$. Then, the process p_E terminates the querying phase and sends back a reply to p_D containing v_E. p_D terminates the querying phase also as its pending list is empty (targets-querying-replied) and sends back the reply containing v_E, v_D. p_A terminates the querying phase, computes the result on the values of all nodes, and returns.

DEPTHSEARCH *correctness proof.* In the following we formally prove that the DEPTHSEARCH algorithm solves the dynamic one time query problem in any model with a bounded but unknown diameter (Theorem 1). In particular, Lemma 1 proves that the DEPTHSEARCH algorithm satisfies Dynamic Validity while Lemma 2 proves that the algorithm satisfies Termination.

Lemma 1 (Dynamic Validity). DEPTHSEARCH *satisfies the Dynamic Validity property in the $M^{\infty,n}$ model.*

Proof. (Sketch) Let G be the graph representing the network when the query starts and let us consider the maximal connected subgraph G' of G at the time the query ends which includes the query initiator p_A. Let us assume by contradiction that when the query ends, p_A does not comprise in its $values$ set the value of one node p_X in the graph G'.

Since p_X belongs to G', then there exists a non-empty set of paths (generally non-independent) which connect p_A and p_X belonging to G' (and G). Without loss of generality let us suppose that there exists only one of such paths $P = \{p_A, ..., p_X\}$.

Let us first observe that when a process p_i receives a query q, the query has actually traversed a sequence of processes which are in the querying state and always comprising the initiator p_A. Let us call this sequence as the query path for the received query. By

construction (line 9) no process in the sequence is in the $replied$ set of any process of the sequence. Then, if p_i replies back to the query, its value starts to flow back on the query path, and each processes which receives it, stores p_i in the $replied$ set and the p_i's local value in the $values$ set. On the other hand, if the query path breaks, then the flowing of the value towards p_A could block. All nodes which did not receive the value back are a prefix of the broken query path and are still in their querying phase (a node is in the querying phase at least the time its succesor in the path is in the querying phase). None of these processes have p_i in the $replied$ set then, a new query path reaching p_i with this prefix is still possible. Moreover, even disconnected nodes which have p_i in the replied set, will renew this set excluding p_i when they receive a new query (line 19) from one of the querying nodes of this prefix.

Let us now consider the case of p_X. p_X will have the path P connecting it to p_A which is up for the whole time interval. However, it could receive a query from another query path which breaks before p_A gets the value of p_X. This could happen more than once, depending on the graph G topology and changes while the algorithm work. Without loss of generality let us suppose that only two paths connect p_A to p_i, i.e. P and a path F which shares with P a non-empty prefix pfx, and the F is the first explored by the algorithm. Let us also suppose that the path F breaks leaving nodes of pfx without the value of p_X. In this case the last node of pfx, let's say p_l, once revealed the disconnection explores another path with the same prefix pfx. Without loss of generality we can now suppose it will explore P. In fact we can assume that all other explored paths before P could complete correclty bringing then the p_l to query its successor in P (by the accuracy property of the failure detector no nodes in the path P can be excluded), this successor does the same as each process between p_l and p_X, leading then to query p_X. By contradiction we assumed that p_X was not in the p_A values when p_A stops to be querying, however when p_A stops to be querying the value of p_X has been surely flowed on the path P leading to a contradiction. $\square_{Lemma\ 1}$

Lemma 2. *The* DEPTHSEARCH *algorithm satisfies* Termination *in the model* $M^{\infty,n}$.

Proof. The only statement blocks the protocol is the wait statement at line 11. Let us call as querying process a process which sent the query message to some node and is waiting for a reply, i.e. a process blocked at statement 11. By the completeness property of the failure detector no querying process can block due to a failure of a node in its neighborhood. Then, let us suppose that no failures happen during the query interval, this also implies that the graph representing the network when the query starts can only grow during this time. By the pseudo-code, a querying node waits a reply from each neighbor which was in the neighborhood when the query is received (line 6). Then, even if in the model $M^{\infty,n}$, a node could have an always growing neighborhood, the neighborhood to wait from never grows, i.e. each querying node p_i has to wait a reply from a bounded number of neighbors n_i. Starting from p_A (the initiator) the query message starts to flow in the graph involving the first neighbor of p_A, which in turn involves its first neighbor and so on. Let us denote as $\{p_1^1, p_1^2, p_1^3 ...\}$ the sequence of processes in which p_1^i is the first neighbor of the process p_1^{i-1}. A first observation on the diameter of the graph which is bounded as the model implies, leads to conclude that this sequence is bounded when all these processes are one different to the other.

On the other hand, since the $querying$ set, sent along with the query, includes all the sequence $\{p_1^1, ...p_1^{i-1}\}$ when arrives at p_1^i, the query stops to flow when (i) either the last process is reached (in the case it contains all different processes) (ii) the first time a process is repeated in the sequence (which means that the sequence contained a loop). Let us denote as p_1^i this last node, it will reply back by letting the process p_1^{i-1} to query its second neighbor. A second observation about the arbitrary order of neighbors in the neighborhoods which make indistinguishable a sequence of processes through where a query flows from another leads to assume $n_i = 1$ for each p_i without lossing generality. This means that each querying process starting from p_1^{i-1} will unblock p_1^{i-2} by replying back its value. All querying processes in the system will eventually unblock preserving Termination. $\square_{Lemma\ 2}$

Theorem 1. *The* DEPTHSEARCH *algorithm solves the dynamic one time query problem in the model* $M^{\infty,n}$.

Proof. It immediately follows from Lemma 1 and Lemma 2. $\square_{Theorem\ 1}$

3.5 Impossibility of Solving the Dynamic One-Time Query Problem in $M^{\infty,\infty}$

This proof is simple. It is based on the race among the message that arrives at a process p_i just a moment before a new process p_{i+1} joins linking to p_i. The race is infinite as a diameter always growing makes possible stretching the path by one infinitely often.

Theorem 2. *The dynamic one-time query problem cannot be solved in the model* $M^{\infty,\infty}$.

Proof. Let us suppose by contradiction that given any operation $query()$, (i) $query()$ will take a finite time Δ, (ii) the operation gathers values from all processes inside the graph for the whole time duration and which are connected to the querying process through the graph defined at the time the query starts or its subgraphs.

Let consider a process p_i invoking a $query()$ operation at some point of time t_q. Let us suppose that a time t_0 (initial time) the network graph consists of a finite path of processes denoted as $\{p_i, \dots p_k\}$. Then, let us suppose that this path infinitely grows along the time, without loss of generality, let us suppose that the path length is increased by 1, by adding a process p_h^i, each δ time interval. Then after $t_0 + n\delta$ the graph consists of the path $\{p_i \dots p_k, p_h^1, p_h^2 \dots p_h^n\}$.

Let us now consider a run R in which $t_q = k\delta + t_0$. In this case all processes $\{p_i \dots p_k, p_h^1, p_h^2 \dots p_h^k\}$ must necessarily receive the query message in order to be involved in the $query()$ operation as the specification requires. This also implies that the process p_h^{k-1} must send the query message to p_h^k. By construction p_h^k belongs to p_h^{k-1}'s neighborhood before the time t_r in which p_h^{k-1} receives the query message.

Now we consider a run R' with the same scenario as R but with $t_q = (k-1)\delta + t_0$ and the time at which p_h^{k-1} receives the query message is again t_r where $t_r > k\delta + t_0$; As p_h^{k-1} cannot determine t_q, then R and R' are indistinguishable for p_h^{k-1}. This means that in R', p_h^{k-1} will relay the message to p_h^k.

This implies that, each process receiving a query message must relay it to the neighborhood defined at the time the query message has been received. Each $query()$ operation can terminate only when a reply has been gathered by all these processes.

Then, consider a run in which each process p_h^i receives the query message at time $t_r^i > (i+1)\delta$. The number of processes will receive the query message will be infinite and Δ is infinite as well, getting a contradiction. $\square_{Theorem\ 2}$

4 Conclusion

The aim of this position paper was the investigation of two attributes that characterize dynamic distributed systems, namely the varying size of the system (according to process joins and departures), and its "geography" captured by the notion of process neighborhood. In order to illustrate these notions, the paper has considered the One-Time query problem as a benchmark problem. It has been shown that (1) the traditional definition of this problem has to be weakened in order the problem can be solved in some dynamic models, and (2) it cannot be solved in all dynamic models. The quest for a general definition of what a "dynamic distributed system" is (a definition on which the distributed system and network communities could agree) still remains a holy grail quest.

References

1. Aguilera, M.K.: A Pleasant Stroll Through the Land of Infinitely Many Creatures. ACM SIGACT News, Distributed Computing Column 35(2), 36–59 (2004)
2. Bawa, M., Gionis, A., Garcia-Molina, H., Motwani, R.: The Price of Validity in Dynamic Networks. In: Proc. ACM Int'l Conference on Management of Data (SIGMOD), pp. 515–526. ACM Press, New York (2004)
3. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
4. Chandra, T., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. Journal of the ACM 43(4), 685–722 (1996)
5. Chandy, K.M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Trans. on Computer Systems 3(1), 63–75 (1985)
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
7. Gafni, E., Merritt, M., Taubenfeld, G.: The concurrency hierarchy, and algorithms for unbounded concurrency. In: Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC '01), pp. 161–16 (2001)
8. Hadzilacos, V., Toueg, S.: Reliable Broadcast and Related Problems. In: Distributed Systems, pp. 97–145. ACM Press, New York (1993)
9. Merritt, M., Taubenfeld, G.: Computing with Infinitely Many Processes. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 164–178. Springer, Heidelberg (2000)
10. Raynal, M.: A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. ACM SIGACT News, Distr. Computing Column 36(1), 53–70 (2005)