From Static Distributed Systems to Dynamic Systems

Achour MOSTEFAOUI⋆ Michel RAYNAL⋆ Corentin TRAVERS⋆
Stacy PATTERSON† Divyakant AGRAWAL† Amr EL ABBADI†

⋆ IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
† Department of Computer Science, University of California, Santa Barbara, CA 93106
{achour|raynal|ctravers}@irisa.fr {sep|agrawal|amr}@cs.ucsb.edu

Abstract

A noteworthy advance in distributed computing is due to the recent development of peer-to-peer systems. These systems are essentially dynamic in the sense that no process can get a global knowledge on the system structure. They mainly allow processes to look up for data that can be dynamically added/suppressed in a permanently evolving set of nodes. Although protocols have been developed for such dynamic systems, to our knowledge, up to date no computation model for dynamic systems has been proposed. Nevertheless, there is a strong demand for the definition of such models as soon as one wants to develop provably correct protocols suited to dynamic systems.

This paper proposes a model for (a class of) dynamic systems. That dynamic model is defined by (1) a parameter (an integer denoted α) and (2) two basic communication abstractions (query-response and persistent reliable broadcast). The new parameter α is a threshold value introduced to capture the liveness part of the system (it is the counterpart of the minimal number of processes that do not crash in a static system). To show the relevance of the model, the paper adapts an eventual leader protocol designed for the static model, and proves that the resulting protocol is correct within the proposed dynamic model. In that sense, the paper has also a methodological flavor, as it shows that simple modifications to existing protocols can allow them to work in dynamic systems.

Key-words: *Communication abstraction, Dynamic system, Persistent reliable broadcast, Query-response pattern, Eventual stability condition, Peer-to-Peer system.*

1 Introduction

Context of the paper P2P systems are evolving rapidly and are becoming a viable paradigm for distributed system computing. Originally, P2P systems were developed to facilitate the sharing of data among a collection of dispersed peers. However, with the increasing popularity of P2P, several proposals have emerged where P2P systems are used as the basis for more complex applications. For example, grid computing on a P2P system would not only exploit available storage, but also potentially harness all available computing and processing power in the Internet to execute different computationally intensive applications. Hence, we can expect P2P systems to evolve into full fledged distributed systems where data is stored and processing is performed in a distributed manner. P2P systems can be viewed as dynamic distributed systems that allow peers to dynamically join and leave the system.

The classical asynchronous message-passing distributed computing model is characterized by the following attributes. The system is made up of n nodes (processes); n is fixed and known by each process; no two processes have the same identity; the whole set of identities is known by each process; the communication network is fully connected; there is no bound on the time it takes for a process to execute a step or for a message to travel from its sender to its destination. A distributed computation in this model is a partial order on the events generated by the execution of the processes [15]. When considering classic distributed systems prone to node failures, another crucial parameter is introduced in the computational model, namely, the maximal number of processes that can be faulty (usually, denoted f). This means that n and f are two fundamental parameters of the classic distributed computing model. From a protocol design point of view, a process can safely use these parameters in its protocol.

It is worth noticing that (since the very early of the eighties) this static model has been questioned by theoreticians interested in the computability power of distributed systems. Their efforts were focused on the following fundamental question [3]: “How much does each processor in a network need to know about its own identity, the identities of other processors, and the underlying connection network in order to be able to carry out useful functions?" This research direction has given rise to notions such as "local knowledge" vs "global knowledge" [6], anonymous networks, sense of direction [9], etc.

The advent of P2P systems (such as [25, 26, 28]) consequently questioned the relevance of the static distributed computing model from a practical point of view. Dynamic systems allow processes (nodes) to dynamically enter and leave the system. It follows that no node can know how many nodes currently constitute the system. The parameters n and f become unknown and meaningless. Roughly speaking, there is no global safe information on the whole system structure that can be used by the nodes.

Motivation of the paper The previous discussion shows that there is a critical need for the definition of distributed computing models for dynamic systems. This is the main issue addressed in this paper whose aim is to propose a computational model for a class of dynamic distributed systems. The development of such models seems crucial for the future, as soon as one wants to understand the basics of dynamic systems, master their difficulty and be able to design provably correct software suited to them[1].

P2P systems suffer from *churn*, namely, that peers join and leave the system at arbitrary times and arbitrarily fast [7, 18, 27]. Churn makes it possible that no node remains long enough in the system for this node (or other nodes) to be able to complete useful computation. So, a dynamic system has to satisfy some form of *eventual stability* in order that a computation be able to progress and terminate. Of course, it is possible that a computation progresses and even terminates during periods where the system is unstable, but no progress guarantee can be associated with such periods. That is why, a system model has to provide *stability conditions* that, when satisfied during a long enough period, ensure that a computation can progress and terminate. (Let us remark that this observation is true even for static systems where the statement "no more than f processes are faulty" is a progress condition provided by the model. If more than f processes do crash, it is possible that no computation at all be able to progress and terminate.)

To capture the notion of eventual stability in a dynamic system (where there is neither the parameter n, nor the parameter f), we consider a new parameter, namely, an integer that we call α. Its value is a requirement on the minimal number of processes that have to be simultaneously alive during a "long enough" period in order for the whole system be able to progress during that period. These α processes are associated with the corresponding period in the sense they constitute -for that period- a *reliable core cluster* able to execute critical tasks or provide basic vital services to the rest of the system. This phenomenon has been observed in P2P systems, including the experimental work of Gummadi et al. [11], where they reported that in unstructured P2P systems, the distribution of average sessions are heavy tailed, i.e., a small number of peers persist for a long time.

It is important to notice that, differently from a static system, "long enough" does not mean "forever". The core cluster of α processes can change over periods according to the processes that enter or leave the system. What is crucial is the fact that providing some vital tasks require the continuous cooperation of α processes. When less than α processes are present, it is possible that the system progresses, but there is no progress guarantees. Actually, α plays the role of the value $n - f$ used in static systems, and that represents a lower bound on the number of processes that are always alive in such systems[2].

When we consider the protocols designed for the static asynchronous distributed computing model (prone to process crashes), we can distinguish at some abstraction level two types of communication primitives used by the processes, namely, a query-response primitive (which can be seen as a generalization of the basic remote procedure call) and a broadcast primitive that allows information dissemination. The first of these primitives is usually expressed by the following sequence of basic statements:

```
broadcast a query message to all the processes;
wait until (responses from $n - f$ proc. have been rec.)
```

while information dissemination is expressed by a simple broadcast invocation:

```
broadcast a msg carrying a new data to all the proc.
```

Let us observe that the implementation of both primitives uses a broadcast. Moreover, the wait statement appearing in the implementation of the query-response primitive uses the value $n - f$ thereby guaranteeing eventual progress (as, due to the model, at least $n - f$ processes are not faulty and can consequently always send responses).

In a dynamic system where n and f "do not exist", each primitive has to be adapted to avoid the dependency on n and f. More specifically, we propose the following communication primitives suited to a dynamic system.

- Query-response pattern.
  In order to ensure that a process that issues a query-response is not blocked forever, we require it to wait

[1]What we call *dynamic/static* systems is sometimes called *open/close* systems in other papers.

[2]A role similar to the α processes is sometimes devoted to the super-peer nodes in some peer-to-peer applications. A notion close to α has been introduced in [14] where the model parameter s is used to denote the smallest number of stable/active processes (a stable process being here a process that is never suspected unless it fails).

for only α responses. This is consistent with the definition of α stated before (see the discussion on eventual stability). Note that the query does not need to be sent to all the processes. More explicitly, the value α in the dynamic model plays the role of the differential value $n - f$ in the static model. This allows getting rid of n and f and replace them by the parameter α related to the eventual stability of the system.

- Information dissemination.
  The specification of a reliable broadcast primitive for a static system is simple: the broadcast message is sent to the n processes, and each message delivered by a process is delivered at least by the correct processes (i.e., the processes that do not crash) [13]. Defining a reliable broadcast primitive in an asynchronous dynamic system requires some care. While the processes that have left the system when a message is sent can be considered as "crashed" (and are consequently not required to deliver the message), this is not the case for the processes that have not yet entered the system. So, the definition of a dynamic system model requires an appropriate reliable broadcast primitive such as the *persistent reliable broadcast* introduced in [10]. Intuitively, such a primitive allows a process to send a message to a "sufficiently large subset" of processes of the dynamic system, "sufficient" meaning that a value that has been broadcast will not be lost by the system considered as a single entity.

It is important to see that the "broadcast to all" used in a static system has to be replaced in a static system by "broadcast to a sufficient number of nodes". The meaning of "sufficient" depends on the particular primitives embedded in the model. Their implementations (that can take into account the underlying overlay structure) is out of the scope of this paper.

Content of the paper The paper presents a model for a class of dynamic systems defined by the value of α, an associated eventual stability condition, a query-response mechanism and a persistent reliable broadcast.

To illustrate the utility of our proposed dynamic systems model, we use leader election as a pedagogical example. A leader election protocol is a basic building block used to solve many synchronization problems that has been extensively studied in static distributed systems. As P2P systems evolve from data centric applications to more general processing applications, the need for such fundamental primitives will increase.

We take a simple leader election protocol for static distributed systems [23], and demonstrate how to adapt it to a dynamic system. The aim is here to illustrate the power and simplicity of our proposal. The modifications are particularly simple: they consists of replacing $n - f$ by α, and using the appropriate dynamic model communication primitives. Of course, the proof of the dynamic protocol has to be adapted to take into account the definition of the dynamic model (mainly the properties of the associated communication primitives, and the eventual stability property). This protocol is exemplary for such an extension since it uses both communication abstractions. This means that other protocols designed for the traditional static model can be easily "translated" to the proposed dynamic model if these protocols only use $n - f$ (instead of explicitly using n or f), a query-response mechanism and a reliable broadcast communication primitive.

2 Dynamic System Model

The system is made up of processes communicating by sending and receiving messages through an underlying network. The first subsection describes the process model, the second subsection describes the communication primitives provided to the processes.

To simplify the presentation, we assume the existence of a discrete global clock. This clock, whose domain denoted $\mathbb{N}$ is the set of integers (plus $+\infty$), is a fictional device that is not known by the processes (which means that this time notion cannot be used by a process inside its protocol).

2.1 Process Model

The system has infinitely many processes but each run has only finitely many. This means that there is no bound on the number of processes for all runs: whatever the integer value n, there are runs with more than n processes. There is a bound on the number of processes in each run, but a protocol does not know that bound because it varies from run to run. This means that no protocol designed for this model can use an upper bound on the number of processes. This is the *finite arrival model* investigated in [1, 19].

Each process has a unique identity. A process knows its identity but does not necessarily know the identities of the other processes. In the following we consider that an identity is a positive integer and p_i will denote the process whose identity is i.

A process enters the system by executing a *join()* operation that provides it with an identity. The processes are asynchronous in the sense that there is no bound on the time needed by a process to execute a computation step. A process can leave the system by executing a *leave()* operation. It can also crash before having executed a leave operation. A process that leaves the system or crashes can re-enter the system with a new identity: it is then considered as a new process. Due to the "finite arrival" assumption, the number of re-entries/recoveries is bounded.

In the following, we use the following notation:

- $up(t)$ denotes the set of processes that are present in the system at time t, $t \in \mathbb{N}$ (i.e., $up(t)$ is the set of processes that joined the system before time t and, if any, the associated crash or leave does not appear before time t).

Dynamic systems are characterized by the succession of unstable periods followed by stable periods. As we have seen in the Introduction, progress can be guaranteed only during stable periods if they last long enough. More precisely, let t_b and t_e be the times at which starts and finishes a period. These time instants are defined by the application processes. Basically, t_b is the beginning of the application and t_e its end. Let an interval I be the period $[t_b, t_e]$. The stable set associated with such an interval I is then defined as follows:

- $STABLE(I) =$ $\{i \mid \exists t \in [t_b, t_e] : \forall t' : \ t \le t' \le t_e : \ p_i \in up(t')\}$.

For simplicity, we consider a single interval in the following. (Without loss of generality, all the definitions can be extended to take into account explicitly defined intervals.) Hence, the revised definition is as follows:

- $STABLE = \{i \mid \exists t : \forall t' \ge t : \ p_i \in up(t')\}$. $STABLE$ is the set of processes that, after having entered the system, neither crash nor leave.

In order for the dynamic system model to allow progress despite process arrivals and departures, a progress condition has to be an integral part of the model. This is the role of the integer α where each value of α defines a model instance (as already noticed α captures the differential value $n - f$ used to prevent processes from blocking forever in the static model)[3]. The progress condition we consider for the model is consequently

$$|STABLE| \ge \alpha.$$

The set $STABLE$ is the counterpart of the set of correct processes appearing in the definition of the static model. In the static model, a process is *correct* in a run if it does not crash during that run. Otherwise, it is *faulty*. So, we have the following correspondence between the traditional static model and the proposed dynamic model:

$$\begin{array}{rll} n - f & \text{corresponds to} & \alpha, \\ p_i \text{ is correct} & \text{corresponds to} & i \in STABLE. \end{array}$$

[3]The determination of an appropriate -realistic- value of the model parameter α poses the same problem as the determination of the parameter f in a classical static system prone to failures (what does happen when more than f processes crash?). That determination depends on both the upper layer application and the specific features on the underlying system. (As far as the value of f in static systems is concerned, the interested reader can consult [20] where appropriate values for f are determined when one is interested in the condition-based consensus problem).

2.2 Communication Model

We assume that, as soon as a process p_i knows the identity of a process p_j (from which we can conclude that p_j has entered the system), p_i can send a message to p_j. If p_j neither leaves the system nor crashes, it eventually delivers the message.

In addition to the traditional point to point communication primitives, the model is equipped with the two communication abstractions presented in the Introduction. This section defines them precisely. As already indicated, their efficient implementation is an important open problem that is not addressed in this paper.

Query-response abstraction A query-response communication instance is started by a process p_i when it invokes the primitive issue_query(m) (where m is the query parameter). By definition, the message m has the "query" type, which allows to determine whether it is issued by a query invocation or by a prst_broadcast() (defined below). This entails the sending of the query m to the processes of the system. When a process p_j receives a message of the "query" type, it systematically answers by sending back a response to the sender of m. When p_i has received responses from α processes it stops waiting and continues its local computation. These first α responses are the *winning* responses for that query.

A query-response instance can be seen as a message exchange pattern initiated by a process and terminating when that process has received responses from enough processes. More formally, a query-response satisfies the following properties.

- QR-Validity. If a query message m is delivered by a process, it has been sent by a process.
- QR-Uniformity. A query message m is delivered at most once by a process.
- QR-Termination. If a process p_i does not crash while it is issuing a query, that query generates at least α responses.

A "brute force" implementation of a query-response pattern can be done by the initiator repeatedly flooding the system with the query message until it has received α responses. More efficient implementations can be envisaged (such implementations can benefit from the techniques developed to implement remote procedure call).

Persistent reliable broadcast The second communication abstraction provided by the model is the *persistent reliable broadcast* primitives (that we have introduced in [10]). It is made up of two communication primitives denoted prst_broadcast() and prst_deliver(). These primitives assume that each message m has a type $type(m)$

and a logical date $ld(m)$ ([4]). When a process executes prst_broadcast(m) (resp., prst_deliver()), we say that it "broadcasts" (resp., "delivers") m. The *persistent reliable broadcast* communication abstraction is defined by the following properties:

- PRB-Validity. If a message m is delivered by a process, it has been broadcast by a process.
- PRB-Uniformity. A message m is delivered at most once by a process.
- PRB-Termination. If (1) the process that broadcasts a message m with logical date ld does not crash while issuing the broadcast, or (2) m is delivered by a process, then any process p_j such as $j \in STABLE$ eventually delivers all the messages m' (of the same type as m) such that $ld(m') = ld(m)$, or a message m' (of the same type as m) such that $ld(m') > ld(m)$.

PRB-validity and PRB-uniformity are *safety* properties. The first states that no spurious message is created, while the second states that no message is duplicated. The last property addresses the *liveness* of message deliveries. The first item states that if a process does not crash during the broadcast of a message, that message is not lost in the sense that it is delivered by at least one process. Due to asynchrony and the fact that processes can crash, or dynamically join/leave the system, it is not possible to require that all the processes that are active when a message m is broadcast will deliver the message. Hence the rationale for the second item that states that if a message is delivered by a process, then all the processes that will remain permanently in the system and neither leave it nor crash (i.e., the processes defining the set denoted $STABLE$), will deliver this message or a message of the same type sent later.

Let us observe that in a static system where all the messages have different types, the type notion disappears and logical dates become useless, the primitives prst_broadcast() and prst_deliver() then boil down to the classical uniform reliable broadcast primitives defined in [13].

An implementation of the *persistent reliable broadcast* abstraction can be done according to the following lines. When a process receives a message m, the process first forwards m to all the other processes, and only then delivers the message to itself (the way message forwarding is ensured depends on the underlying overlay network and the associated routing [25, 18, 26, 28]). Moreover, a new process that joins the system has first to broadcast (using the underlying routing) an inquiry message to the processes currently present in the system. When a process receives such a message, it sends back its state and, for each message type, the logical date of the last message it has delivered.

[4]A type is a *tag* that allows associating an appropriate processing with each message. As defined by Lamport in [15], the logical dates, managed by the application processes according to the problem they solve, are monotonically increasing (see Section 4.1 for an example).

3 Eventual Leader in a Dynamic System

As announced in the introduction, this section can to be considered as a "proof by example": to motivate the previous dynamic system model, we show how a protocol designed for a static system can be extended to work in a dynamic system. This section also has a methodological flavor: it shows that when one has to solve some problems in a dynamic system, it is not always necessary to design a new protocol from scratch, a simple adaptation of a protocol initially designed for the static model can provide a corresponding protocol for the dynamic model. According to these observations, we consider here a non-trivial distributed computing problem, namely, the election of an eventual leader. Leader election, in itself is an fundamental problem, whose solution is important for solving many synchronization problems. We expect that P2P systems will need to address such problems as their domain of applications increases.

3.1 Problem Definition

A *leader* oracle is a distributed entity that provides the processes with a function leader() that returns a process name each time it is invoked. A unique leader is eventually elected but there is no knowledge of when the leader is elected. Several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when this "anarchy" period is over. Moreover, (to be useful) the eventual unique leader p_ℓ is such that $\ell \in STABLE$. The *leader* oracle (denoted Ω) satisfies the following property:

- Eventual Leadership: There is a time t and a process p_ℓ such that $\ell \in STABLE$, and, after t, every invocation of leader() by any process returns ℓ.

This definition boils down to the the traditional definition of eventual leader used in static systems when we replace the set $STABLE$ by the set of correct processes.

3.2 Eventual Leader in a Static System

Using a leader in a static system Several protocols suited to static systems are based on an eventual leader oracle. Among them, Ω-based consensus algorithms are the most known. Such protocols are described in [12, 16, 22][5] for

[5]The Paxos protocol [16] is leader-based and considers a more general model where processes can crash and recover, and links are fair lossy. (Its first version dates back to 1989, i.e., before the Ω formalism was introduced.)

systems where a majority of processes are correct ($f < n/2$). Such consensus algorithms can then be used as a subroutine to implement atomic broadcast protocols (e.g., [4, 16]). (When we consider the failure detector-based approach to solve consensus [4, 24], it has also been shown that an eventual leader facility is the weakest failure detector that can be used to solve that problem in static systems where $f < n/2$ [5].)

Electing a leader in a static system Unfortunately there is no protocol to elect an eventual leader in a static asynchronous system made up of n processes among which up to t can crash. Intuitively, this comes from the impossibility to solve the consensus problem in such systems [8][6].

The impossibility to elect an eventual leader in a fully asynchronous system has motivated researchers to find additional assumptions that, when satisfied by the underlying asynchronous static system, allow implementing an eventual leader. Two such approaches have been investigated.

- One approach consists of enriching the system model with eventual synchrony assumptions on process speed and message delays [2, 17]. Basically, the protocols based on such an assumption ensure that, if eventually the system behaves synchronously, an eventual leader can be elected.

- Another approach consists of enriching the system with an assumption on the message exchange pattern [21]. Basically, it is possible to design a query-response based protocol such that, if the messages generated by the query-response invocations eventually satisfy some pattern, then an eventual leader can be elected. This is the approach we consider here.

3.3 Eventual Leader in a Dynamic System

The additional assumption for a static system Let us consider a process p_i that issues a sequence of queries (p_i starts a new query only when the previous one is terminated, i.e., it has received the $n - f$ responses that complete the previous query). Let us call *winning* a response that arrives among the $n - f$ responses p_i is waiting for. Let $winning_i(t)$ be the identities of the processes from which p_i has received a winning response to its last query terminated before or at t.

The additional assumption (that we call MP_{static}) investigated in [23] to design an eventual leader protocol in a static system is the following:

[6]This is because an eventual leader oracle is the weakest failure detector that allows designing consensus protocols. A direct proof (i.e., a proof that is not based on the consensus impossibility) for the impossibility of implementing a leader can be found in [23].

There are a time t, a correct process p_ℓ and a set Q (t, p_ℓ and Q are not known in advance) such that, $\forall t' \geq t$, we have[7]

1. $|Q| = f + 1$, and
2. $\ell \in \left(\bigcap_{j \in Q \cap up(t')} winning_j(t')\right)$.

The intuition that underlies this property is the following. Even if the system never behaves synchronously during a long enough period, it is possible that its behavior has some "regularity" that can be exploited to determine an eventual leader. This regularity can be seen as some "logical synchrony" (as opposed to "physical" synchrony). More precisely, MP_{static} states that, eventually, there is a cluster Q of $(f + 1)$ processes (item 1) such that each of them (until it possibly crashes) receive winning responses from p_ℓ to all its queries (item 2). This can be interpreted as follows: among the n processes, there is a process that has $(f + 1)$ "favorite neighbors" with which it communicates faster than with the other processes.

The underlying idea exploited in the eventual leader protocol introduced in [23] for static systems is the following.

- First, the set Q of $f + 1$ processes always includes at least one correct process. We can then conclude that if, each time it issues a query, a process waits for responses from $n - f$ processes, it always receives a message from at least one process of Q.

- Due to the additional assumption MP_{static}, there is a time after which the non-crashed processes of Q never "suspect" the process p_ℓ (because they always receive a winning response from it). So, if processes use a gossiping mechanism to exchange the identities of the processes they do not suspect, after some time no process will suspect p_ℓ.

- Finally, each process elects as a leader the process with the smallest identity among the processes it does not suspect.

Translating the additional assumption to a dynamic system As indicated, the protocol for the static model is based on the fact that any set of $f + 1$ processes always includes at least one correct process, and any set of $n - f$ processes always intersect any set of $f + 1$ processes. So, $f + 1$ and $n - f$ are critical values for the static protocol.

Let us consider a process p_i that issues a sequence of queries (p_i starts a new query only when it has received α responses to the previous one). As before, we say that a response is *winning* if it arrives among the α responses p_i

[7]In the static model, $up(t')$ denotes the set of processes that have not crashed at time t'. The static model (without process recovery) is characterized by the following monotonicity property $t1 < t2 \Rightarrow up(t2) \subseteq up(t1)$. This property is is no longer satisfied in a dynamic system.

is waiting for. $winning_i(t)$ has exactly the same meaning: it is the set of the identities of processes from which p_i has received a winning response to its last query terminated at or before t. The static additional assumption MP_{static} can be translated as follows for a dynamic system. This new formulation is denoted MP_{dyn}.

There are a time t, a process p_ℓ such that $\ell \in STABLE$, and a set Q of processes (t, p_ℓ and Q are not known in advance) such that, $\forall t' \geq t$, we have

1. $Q \subseteq up(t')$, and
2. $\ell \in \big(\cap_{j \in Q}\ winning_j(t')\big)$, and
3. $\forall x \in up(t') : \ Q \cap\ winning_x(t') \neq \emptyset$.

The items 1 and 2 correspond to the items of the static model: they state that there is a set Q of processes that (after some time) never "suspect" the same process p_ℓ (because they always receive winning responses from it). The last item states that, after t', it is possible for each process to get information from at least one process in Q. It correspond to the implicit item of the static model saying that any set of $f+1$ processes intersect any set of $n-f$ processes. These three items constitute a translation of MP_{static} that works in a dynamic system. The next section shows how an MP_{static}-based protocol can be modified to get an MP_{dyn}-based protocol.

4 A Leader Protocol for a Dynamic System

Two MP_{static}-based eventual leader protocols are described in [23]. The first uses an array with one entry per process. Clearly, due to this array structure, this protocol is not suited to the dynamic model. So, we consider the second of these protocols whose basic data structure is a set of process identities. In the following $\top$ denotes the whole universe of possible processes (e.g., 128-bit identifier space for peers in Chord [26]). Moreover, $\top \cap A = A$ (where A is any set of processes).

4.1 Description of the Protocol

The protocol is described in Figure 1. With respect to the static protocol [23] that it extends, the lines that are new or modified are prefixed by a "$\star$" in Figure 1. This protocol is made up of 4 tasks. The aim is for each process p_i to maintain a set of process identities, denoted $trust_i$, such that eventually all these sets have the same value, thereby allowing each process p_i to elect the same process from its set $trust_i$. This is role of the task $T4$.

The task $T1$ is the core task in which each process initiates sequential queries and waits for the corresponding responses. The task $T2$ implements the response mechanism associated with the queries: when p_i receives a query, it sends back a response carrying a value (line 7). The task $T3$ describes the processing of a message sent by a persistent broadcast invocation.

In addition to the set $trust_i$, a process p_i manages two additional local data structures, namely, a set rec_from_i and an integer log_date_i. rec_from_i contains the identities of the α processes that sent winning responses to the last query issued by p_i (line 5). log_date_i is a logical date defining the "age" of set $trust_i$ (line 8). The time during which log_date_i keeps the same value is called an "epoch".

More explicitly, each process p_i repeatedly issues a query and waits for α responses[8]. This allows it to compute the last value of the set rec_from_i. The response from p_j carries the last value of rec_from_j. Hence, from the rec_from_j sets it has received, p_i determines the set of processes it currently trusts (line 4). To inform the other processes of its new $trust_i$ set, the process p_i uses gossiping: it invokes the persistent reliable broadcast to disseminate the new pair defining its current state, namely, $(trust_i, log\_date_i)$ (line 6). The type of a message $\text{TRUST}(trust, log\_date)$ is TRUST and its logical date is the value of log_date.

When p_i receives a message $\text{TRUST}(trust_j, log\_date_j)$ sent by p_j with the persistent reliable broadcast primitive, it updates $trust_i$ according to the respective value of log_date_j and log_date_i. If they are equal, p_i considers their intersection as the new value of the set of processes it trusts (line 8). If its current knowledge is too old (i.e., $log\_date_i < log\_date_j$), it adopts the values received (line 9). Otherwise, it discards the message received. If, after these updates, its set $trust_i$ is empty, p_i starts a new "epoch" by increasing log_date_i and resetting $trust_i$ to its initial value (line 10). The age of this new epoch is the new value of log_date_i. Note that during an "epoch", a set $trust_i$ can only remain constant or decrease.

The proof will show that there is an epoch with a finite age after which the protocol "stabilizes", i.e., the log_date_i values no longer increase and the sets $trust_i$ of the processes currently in the system are (and remain) non-empty and equal. They actually converge to a subset of the $STABLE$ set. The process in these $trust_i$ sets with the smallest identity is then elected as the unique leader.

4.2 Proof of the Protocol

The proof has the same structure as the proof of the static protocol. It differs from it in the way it takes into account the new assumption associated with a dynamic system. It

[8]The duration between two consecutive executions of the "repeat" statement is arbitrary. Its activation has to seen seen as being event-based and not timeout-based.

```
⋆ init: $rec\_from_i \leftarrow \top$; $log\_date_i \leftarrow 0$; $trust_i \leftarrow \top$;

  task $T1$:
    repeat
⋆ (1)   issue_query QUERY$(i)$;
⋆ (2)   wait_until ( corresponding RESPONSE$(rec\_from)$ received from $\alpha$ processes );
  (3)   let $REC\_FROM_i = \cup$ of all the $rec\_from_k$ received at line 2;
  (4)   $trust_i \leftarrow trust_i \cap REC\_FROM_i$;
  (5)   $rec\_from_i \leftarrow$ the set of processes from which $p_i$ received a RESPONSE at line 2;
⋆ (6)   if ($trust_i$ modified at line 4) then prst_broadcast TRUST$(trust_i, log\_date_i)$ endif
    end_repeat

⋆ task $T2$: when QUERY$(j)$ is delivered:
⋆ (7)   send RESPONSE$(rec\_from_i)$ to $p_j$

  task $T3$: when TRUST$(trust_j, log\_date_j)$ is delivered:
  (8)   if $log\_date_j = log\_date_i$ then $trust_i \leftarrow trust_i \cap trust_j$ end_if;
  (9)   if $log\_date_j > log\_date_i$ then $trust_i \leftarrow trust_j$; $log\_date_i \leftarrow log\_date_j$ end_if;
  (10)  if $trust_i = \emptyset$ then $trust_i \leftarrow \top$; $log\_date_i \leftarrow log\_date_i + 1$ end_if

  task $T4$: when leader() is invoked by the upper layer:
⋆ (11)  if $trust_i = \emptyset \vee trust_i = \top$ then return $(i)$
  (12)                                   else return $(\min(trust_i))$
  (13)  end_if
```

Figure 1. An eventual leader protocol for a dynamic system (code for process p_i)

addresses the "worst case" scenario where a leader can only be elected when the eventual progress assumption (defined by the set $STABLE$) becomes satisfied. From a practical point of view, it is important to notice that the protocol has runs where an eventual leader is elected before that assumption becomes satisfied.

Preliminaries We use the following notation in the proof: $x_i(t)$ denotes the value of the local variable x of p_i a time t.

The model assumes that there eventually exists a set $STABLE$ of processes that, after having entered the system, neither crash nor leave. In the following, we say that a process p_i is *stable* if $i \in STABLE$. Let us remark that the set $STABLE$ is not empty (the progress condition states that $|STABLE| \geq \alpha > 0$). Moreover, due to the finite arrival assumption, there exists a time, say τ_0, such that, after τ_0, no more processes join the system. Consequently, there is a time $\tau \geq \tau_0$ such that (1) every process that belongs to the $STABLE$ set has entered the system before τ and, (2) every process that does not belong to the $STABLE$ set has left or crashed before τ. For our purpose, namely, to prove that there is a time after which a stable process is elected as a common leader, we consider the time instants t such that $t \geq \tau$.

Let us notice that, as a query invocation generates at least α RESPONSE messages (QR-Termination property) and due to the existence of the $STABLE$ set (which size is at least α), no process can be blocked forever in the **wait** statement at line 2.

Lemma 1 *There is a time t and a stable process p_ℓ (i.e., $\ell \in STABLE$) such that every REC_FROM set computed (at line 3) after t is such that $\ell \in REC\_FROM$.*

Proof Given an execution that satisfies the MP_{dyn} assumption, there is a time $t0$ after which there is a a stable set Q and a stable process p_ℓ such that $t \geq t0 \wedge i \in Q \Rightarrow \ell \in rec\_from_i(t)$. Moreover, due to the intersecting property of the MP_{dyn} assumption, a process p_j that starts a query (at line 1) after $t0$ obtains (if p_j does not crash while the query is on progress) a winning response from at least one process p_i such that $i \in Q$ (i.e., $rec\_from_j \cap Q \neq \emptyset$). More, the rec_from set carried by the RESPONSE message from p_i is such that $\ell \in rec\_from$ (this is because the query considered starts after $t0$). It follows from the way REC_FROM is computed (line 3) that $\ell \in REC\_FROM_j$. This discussion implies that for every stable process p_j, there exists a time t_j after which ℓ continuously belongs to REC_FROM_j (t_j is the time at which p_j computes its REC_FROM set and, that immediately follows the first query invocation issued by p_j started after $t0$)

Let $t1 = \max(t0, \tau)$, where τ is the time defined in the preliminary (i.e., $\forall t \geq \tau : up(t) = STABLE$). More, let $t2 \geq t1$ be the first time after which every process in $up(t1)$ has started and terminated a query. More precisely, $t2 = \max\{t_j : j \in up(t1)\}$. As, after t_j, ℓ continuously belongs to REC_FROM_j, taking $t = t2$ proves the lemma. $\Box_{Lemma\ 1}$

Lemma 2 *$\exists M, \exists t$ such that, $\forall i, \forall t' \geq t : i \in up(t') \Rightarrow log\_date_i(t') = M$.*

Proof Let us define $\tau'(\geq \tau)$ as a time after which no message that has been broadcast (at line 6) before τ (the time defined in the preliminaries) is delivered by any stable process. Such a time exists due to the PRB-Termination property. Moreover, let $t0 = \max(\tau', t)$ where t is the time defined in Lemma 1. Notice that $\exists \ell \in STABLE$ such that any REC_FROM set computed after $t0$ contains ℓ.

Let m_log_date be the maximum logical date among the stable processes at time $t0$. Moreover, let say "the set $trusted$ is associated with the logical date ld" when there is a stable process p_i and a time $t \geq t0$ such that $log\_date_i(t) = ld$ and $trust_i(t) = trusted$. Let us remark that several sets can be associated with the same logical date ld.

Claim C. Let us assume that $\emptyset$ is associated with m_log_date. There is then (1) a process p_j that executes the reset statement at line 10, after which we have $(trust_j, log\_date_j) = (\top, m\_log\_date + 1)$. Moreover, (2) $log\_date_j(t) = m\_log\_date + 1 \Rightarrow \ell \in trust_j(\ell)$.

Proof of the claim. Let us first observe that (Observation $O1$) a set $trust_i$ can only decrease while log_date_i remains equal to m_log_date, (Observation $O2$) there is no gap in logical dates (which means that if a logical date variable is equal to m, then there are logical date variables that had previously the values $0, 1, \ldots, m-1$), and (Observation $O3$) the update by a process p_j of its log_date_j variable to the value $m\_log\_date + 1$ (at line 9 or 10) is always due to the fact that some stable process p_k (which is possibly p_j itself) executed $log\_date_k \leftarrow log\_date_k + 1$ at line 10 (where m_log_date is the value of log_date_k before the update; notice that p_k also sets $trust_k$ to $\top$).

Let p_i be a process that associates $\emptyset$ with m_log_date. If the pair $(trust_i, log\_date_i)$ remains equal to $(\emptyset, m\_log\_date)$ until p_i receives a TRUST message, it executes line 10 and resets $(trust_i, log\_date_i)$ to $(\top, m\_log\_date + 1)$. The only other possibility for that pair to be modified is at line 9, but in that case p_i received a logical date $> m\_log\_date$, and it follows from the observations $O2$ and $O3$ that some process p_j has executed line 10 updating the pair $(trust_j, log\_date_j)$ to $(\top, m\_log\_date + 1)$. This proves the first part of the claim.

The proof of the second part of the claim is by contradiction. Let us assume that a process p_i is such that $\ell \notin trust_i$ and $log\_date_i = m\_log\_date + 1$ at time t_i. Let us observe that (A) the logical date $m\_log\_date + 1$ is introduced in the system at time $t \geq t0$ and, (B) after $t0$, p_i is always such that $\ell \in REC\_FROM_i$ (Lemma 1). Consequently, p_i cannot remove ℓ from its $trust_i$ set at line 4. This means that p_i computes a $trust_i$ that does not contain ℓ while executing task $T3$. As a process that executes the reset statement at line 10 sets its $trust_i$ to $\top$ (which contains ℓ), the only possibility for p_i to be such that $\ell \notin trust_i$ is to receive a message TRUST $(trusted, m\_log\_date + 1)$ such that $\ell \notin trusted$ from a process p_{i1}. As we can apply exactly the same reasoning to p_{i1}, we conclude that it exists an infinite chain of distinct processes $p_{i0}(= p_i), p_{i1}, p_{i2}, \ldots$ that has broadcast a pair $(trusted, m\_log\_date + 1)$ such that $\ell \notin trusted$ between times $t0$ and t_i. As, due to the finite arrival model, only finitely many processes take finitely many steps during any finite time interval, this is clearly impossible. *End of the proof of the Claim C.*

We show that $M = m\_log\_date$ or $M = m\_log\_date + 1$. We consider two cases:

- Case 1. $\emptyset$ is never associated with m_log_date. In that case, no stable process will ever execute the reset statement at line 10. It follows that no process p_i will increase its log_date_i variable. Due to the definition of m_log_date, there is a stable process p_j such that $log\_date_j(t0) = m\_log\_date$. As a stable process does not block while issuing a query, p_j broadcast a TRUST message that carries m_log_date. As no logical date $ld > m\_log\_date$ is ever generated, it follows from the PRB-Termination property that every stable process receives a message TRUST$(_, m_log_date)$. Consequently, due to the way a process updates it log_date variable (lines 8 or 9), it exists a time after which all processes p_i are such that $log\_date_i = m\_log\_date = M$.
- Case 2. $\emptyset$ is associated with m_log_date. Due to the second part of the Claim *C*, no process p_x can be such that $(trust_x, log\_date_x) = (\emptyset, m\_log\_date + 1)$. Consequently no logical date $> m\_log\_date + 1$ can ever be generated (1). Moreover, due to the first part of the Claim *C*, there is a stable process p_j that executes the reset statement (at line 10), after which we have $(trust_j, log\_date_j) = (\top, m\_log\_date + 1)$. If p_j starts a query invocation with its $trust_i = \top$ then, $trust_i$ is modified at line 4 and consequently p_j broadcasts a TRUST message that carries the logical date $m\_log\_date + 1$. If $trust_j$ is modified before p_j starts a query then, p_j has received a TRUST $(trusted, ld)$ such that $ld \geq m\_log\_date + 1$. As no logical date $> m\_log\_date + 1$ can be generated, we have $ld = m\_log\_date + 1$ which means that a process has broadcast a TRUST message that carries $m\_log\_date + 1$. It follows then from (1) and the PRB-Termination property that such a message is eventually delivered by all the stable processes. Consequently, there is a time after which every process p_i is such that $log\_date_i = m\_log\_date + 1 = M$. $\Box_{Lemma\ 2}$

Theorem 1 *The protocol described in Figure 1 implements a leader facility in a dynamic system.*

Proof Given a run, let: $PL = \{x \mid \exists i \in STABLE :$ after some time x remains continuously in $trust_i\}$.

We first show that $MP_{dyn} \Rightarrow PL \neq \emptyset$. This is a consequence of Lemma 2 (which relies on the MP_{dyn} assumption). More precisely, there is a time t after which every process in the system has the same logical date M and this logical date does not increase, from which we conclude that no set $trust_i$ becomes empty after t. Moreover, due to the PRB-Termination property and as the sequence numbers do not increase, any $trusted$ set that is broadcast after t is delivered by every stable process. This ensures that there is a time $t' \geq t$ after which all these sets are equal and do not change their value (as they are then updated only by intersection). Finally, due to the very definition of PL, the $trust_i$ sets are then equal to PL. It follows that PL is not empty.

We now show that $PL \subseteq STABLE$. This a direct consequence of the query mechanism used to update $trust$ sets. Let p_x be a process such that $x \notin STABLE$. Since p_x crashes or leaves the system, there is a time after which x does not appear any longer in any rec_from set. Consequently, there is a time after which every REC_FROM does not contain x. Therefore, there is a time after which the REC_FROM_i sets contain only stable processes. Moreover, as the $trust_i$ sets are never reset to $\top$, it follows that, after that time, these $trust_i$ sets can contain only stable processes.

Finally, the eventual leadership property follows from the following observations: $PL \neq \emptyset$, $PL \subseteq STABLE$, and to the fact that, due to the gossiping mechanism (line 6 and task $T3$), there is a time t after which we have $\forall i : i \in STABLE \Rightarrow trust_i = PL$. $\Box_{Theorem\ 1}$

Acknowledgments

The authors would like to thank Gérard le Lann for interesting discussions on distributed computing models.

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] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. *Proc. 23th ACM Symp. on Principles of Distributed Computing*, ACM Press, pp. 328-337, 2004.

[3] Angluin D., Local and Global Properties in Networks of Processes. *Proc. 12th ACM Symp. on Theory of Computing (STOC'80)*, ACM Press, pp. 82-93, 1980.

[4] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *J. of the ACM*, 43(2):23-267, 1996.

[5] Chandra T.D., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *J. of the ACM*, 43(4):685-722, 1996.

[6] Chandy K.M. and Misra J., How Processes Learn. *Distributed Computing*, 1(1):40-52, 1986.

[7] Chawathe Y., Ratnasamy S., Breslau L., Nick Lanham, Shenker S., Making Gnutella-like P2P Systems Scalable. *Proc. of the 2003 Conf. On Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*, pp. 407-418.

[8] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[9] Flocchini P., Mans B. and Santoro N., Sense of Direction in Distributed Computing. *Proc. 12th Int'l Symp. on Distributed Computing (DISC'98)*, Springer-Verlag LNCS #1499, pp. 1-16, 1998.

[10] Friedman R., Raynal M. and Travers C., Two Abstractions for Implementing Atomic Objects in Dynamic Systems. Brief Announcement, *Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, ACM Press, July 2005. (http://www.irisa.fr/bibli/publi/pi/2005/1692/1692.html).

[11] Gummadi K., Dunn R., Saroiu S., Gribble S., Levy H., and Zahorjan J., Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload, *Proceedings of the 19th ACM Symp. on Operating Systems Principles (SOSP'03)*, pp. 381-394, 2003.

[12] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Trans. on Computers,* 53(4):453-466, 2004.

[13] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press, N-Y, pp. 97-145, 1993.

[14] Hermant J.-F. and Le Lann G., Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems. *IEEE Trans. on Computers,* 51(8):931-944, 2002.

[15] Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558-565, 1978.

[16] Lamport L., The Part-Time Parliament. *ACM Trans. on Computer Systems*, 16(2):133-169, 1998.

[17] Larrea M., Fernández A. and Arèvalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symp. on Reliable Distributed Systems (SRDS'00)*, IEEE Computer Society Press, pp. 52-60, 2000.

[18] Liben-Nowell D., Balakkrishnan H. and Karger D., Analysis of the Evolution of Peer-to-Peer Systems. *Proc. 21th ACM Symp. on Principles of Distributed Computing*, ACM Press, pp. 233-242, 2002.

[19] Merritt M. and Taubenfeld G., Computing Using Infinitely Many Processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 164-178, 2000.

[20] Mostefaoui A., Mourgaya E., Raipin Parvedy Ph. and Raynal M., Evaluating the Condition-Based Approach to Solve Consensus. *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 541-550, 2003.

[21] Mostefaoui A., Mourgaya E., and Raynal M., Asynchronous Implementation of Failure Detectors. *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360, San Francisco (CA), 2003.

[22] Mostefaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.

[23] Mostefaoui A., Raynal M. and Travers C., Crash-Resilient Time-Free Eventual Leadership. *Proc. 23th Symposium on Reliable Distributed Systems (SRDS'04)*, IEEE Computer Society Press, pp. 208-217, 2004.

[24] Raynal M., A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. *ACM SIGACT News, Distributed Computing Column*, 36(1):53-70, 2005.

[25] Rowstron A. and Druschel P., Pastry: Scalable, Distributed Object Location and Routing for Large Scale Peer-to-Peer Systems. *Proc.18th IFIP/ACM Int'l Conf. on Dist. Systems Platforms (Middleware 2001)*, Springer-Verlag LNCS #2218, pp. 329-350, 2001.

[26] Stoica I., Morris R., Liben-Nowell D., Karger D., Kaashoek M.F., Dabek F. and Balakrishnan H., Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *ACM/IEEE Trans. on Networking*, 11(1):17-32, 2003.

[27] Stutzbach D. and Rejaie R., Towards a better Understanding of Churn in Peer-to-Peer Networks. *Tech Report CIS-TR-04-06*, Dpt of CS, University of Oregon, 2004.

[28] Zhao B., Kubiatowicz J. and Joseph A., Tapestry: An Infrastructure for Fault-Tolerant Wide-area Location and Routing. *Tech Report UCB/CSD-01-1141*, CS Division, U.C. Berkeley, 2001.