The Eventual Leadership in Dynamic Mobile Networking Environments

Jiannong Cao[1], Michel Raynal[2], Corentin Travers[2], Weigang Wu[1]
1. Department of Computing, The Hong Kong Polytechnic University
{cjcao, cswgwu} @comp.polyu.edu.hk
2. IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{raynal, ctravers}@irisa.fr

Abstract

Eventual leadership has been identified as a basic building block to solve synchronization or coordination problems in distributed computing systems. However, it is a challenging task to implement the eventual leadership facility, especially in dynamic distributed systems, where the global system structure is unknown to the processes and can vary over time. This paper studies the implementation of a leadership facility in infrastructured mobile networks, where an unbounded set of mobile hosts arbitrarily move in the area covered by fixed mobile support stations. Mobile hosts can crash and suffer from disconnections. We develop an eventual leadership protocol based on a time-free approach. The mobile support stations exchange queries and responses on behalf of mobile hosts. With assumptions on the message exchange flow, a correct mobile host is eventually elected as the unique leader. Since no time property is assumed on the communication channels, the proposed protocol is especially effective and efficient in mobile environments, where time-based properties are difficult to satisfy due to the dynamics of the network.

1. Introduction

In asynchronous distributed systems, there is no bound on the time for a process to execute a computation step, or for a message to be delivered. Due to such timing uncertainty, solving coordination problems, e.g. *consensus* and *mutual exclusion* [10], is a difficult and complex task. For example, the consensus problem [10] has been proved to be impossible to solve in an asynchronous system with even one crash failure [17].

To overcome the difficulty introduced by timing uncertainty and process crashes, the concept of *unreliable failure detector* has been introduced [9]. A failure detector can be viewed as an *oracle* [32] made up of a set of modules, each associated with a process. The failure detector attached to a process provides hints on the status (alive or crashed) of other processes. A failure detector is defined by abstract properties and does not depend on any particular assumption on the behavior of the underlying network. Among different failure detectors defined in [9], the *eventual leader*, denoted by Ω, is one of the most important classes. An Ω leader provides the processes with a *leader primitive* that outputs a process *id* each time it is invoked and satisfies the following *eventual leadership* property:

> *Eventual leadership*: eventually, all invocations return the same *id*, and that *id* is the identity of a correct process (i.e. a process that does not crash during the execution of the protocol).

Ω is not very powerful in terms of the capability of detecting failures, since a correct leader is eventually elected but there is no knowledge on when this occurs. However, it has been shown that Ω is the weakest class of failure detectors that allows solving the consensus problem (provided that a majority of correct processes) [10]. Based on Ω, many consensus protocols [10, 21, 29] have been proposed. Ω is also at the heart of the well-known Paxos algorithm [23] and its improvements [14, 20, 22] to cope with dynamic systems.

A large number of researches [3, 4, 9, 12, 24, 26] have been conducted to implement the oracle Ω in a classical asynchronous distributed computing model, which is characterized by the following attributes. The system is made up of n processes and n is fixed and known by each process; each process has a unique identity and knows the identities of other processes; 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.

In recent years, a major advance in distributed computing is the development of dynamic systems [2, 27, 18, 31], e.g. mobile computing systems and peer-to-peer systems, where processes can join or leave the

0-7695-3054-0/07 $25.00 © 2007 IEEE
DOI 10.1109/PRDC.2007.27

system at any time and the number of participating processes can change arbitrarily as time passes. The inherent dynamic nature of processes introduces a new kind of uncertainty, namely *structure uncertainty*: the global structure of the network is unknown to the processes. This additional difficulty makes the design of coordination protocols even more challenging than in classical distributed systems.

This paper investigates the implementation of Ω in dynamic mobile networking environments with mobile hosts[1] (MHs for short) and mobile support stations (MSSs for short). MHs, which are usually small devices with low computation power and stand alone energy sources, are connected to MSSs using wireless communications [7, 8]. Due to mobility, an MH can change its location arbitrarily and enter or leave the area covered by the MSSs. Moreover, to save energy, an MH may voluntarily disconnect from the network. This means that at any time, the mobile processes that form the system are unknown to MHs and MSSs.

To implement Ω in a dynamic mobile network, we adopt a time-free approach [28, 30] proposed for traditional fixed networks and extend it to the context of mobile networking environments. We let MSSs act as servers that provide an eventual leadership service to the MHs. More precisely, MSSs conduct the exchange of queries and responses using the query-response mechanism in [28, 30], in order to elect an eventual unique leader MH (it is important to notice that an MH rather than an MSS can be elected as the leader, because the leadership is for upper layer applications at MHs. MSSs are usually owned by network operators and cannot participate in the execution of end user applications). Such a treatment can reduce the workload of MHs and consumption of various resources, e.g. battery power and bandwidth.

However, with such a design, the query-response mechanism in [28, 30] cannot be directly used. First, the eventual leader is an MH but it is elected by MSSs, so the MSSs must be provided a view of MHs. To do so, we assume that each MSS is equipped with a device/module that provides it with partial information about the MHs that are present in the system. More precisely, each MSS b_i is provided with a set $local\_trust_i$ of mobile process identities that represents b_i's current view of the MHs that are currently present in the system.

Another problem is that the assumption in [28, 30] becomes unreasonable in the mobile system. MHs may move from one cell to another, so it is impossible that, after some finite time, an MH is always accessible by some specific set of MSS. Therefore, the assumption of the host p and set Q does not make sense any more.

To address these problems, we develop a two-phase query-response mechanism. With some additional assumption on the *local_trust*, we implement Ω in a dynamic mobile system. Since no assumption is related to the time property of the message passing channel, the proposed protocol is time-free, so it is especially suitable for mobile networks, where time properties are difficult to satisfy.

The rest of the paper is organized as follows. Section 2 reviews existing work on the implementation of the eventual leader facility Ω. Section 3 presents the system model, i.e. the dynamic mobile networking environment. In Section 4, we describe the formal definition of the eventual leadership with respect to the system model, and the additional assumption MP_{dyn}. Our proposed eventual leadership protocol and the proof of its correctness are presented in Section 5. Finally, Section 6 concludes the paper.

2. Related work

The implementation of the oracle Ω in static/classical asynchronous systems has motivated a large body of researches [3, 4, 9, 12, 24, 26]. The first approach investigated in [9, 12, 24] considers that all the links connecting the processes are eventually timely. This means that after some time τ, each message reaches its destination in at most δ units of time. Both τ and δ are unknown to the processes [13]. This approach has been refined to obtain weaker constraints. It has been shown in [3, 4] that Ω can be implemented in a system where at least one correct process has at least s eventually timely outgoing links (this is defined as s-source).

Interestingly, a step ahead has been taken in [26], where the notion of eventual s-accessibility is introduced. Informally, a process p is s-accessible at some time if messages sent by p at that time are received within δ units of time by a set Q^{τ} of at least s processes. The interest of this notion lies in the fact that the set Q of processes that "witness" p can be different at different time.

A time-free approach [28, 30] implements the eventual leadership using a query-response based mechanism. There are totally n processes in the system and at most t of them can fail (by crashing only). The solutions in [28, 30] rely on an assumption on the behavior of the flow of message exchange. More precisely, processes broadcast queries and then wait for responses from other processes. The first n-t responses received are *winning* responses (the other responses, if any, are called *losing* responses; they can be slow or

1 In this paper, we use the terms "process" and "host" interchangeably.

never sent due to the crash of the sender). It is shown in [30] that Ω can be built if the following behavioral property is satisfied: "there is a correct process *p* and a set *Q* of $t + 1$ processes such that eventually each response of *p* to each query issued by any $q \in Q$ is always a winning response". Intuitively, this means that for $q \in Q$, the link connecting *q* to *p* is not among the *t* slowest links of *q*.

Another approach investigated in [6, 11] considers reducing Ω to other failure detector classes. This approach is mainly theoretical: it aims at comparing and ranking different failure detector classes.

In the context of dynamic systems, little work has been done for the implementation of the eventual leadership. Friedman et al. [19] evaluate the gossip based failure detection in mobile ad hoc networks. An eventual leadership protocol is proposed in [31], but the proposed protocol can be viewed as a pedagogical example and is not suitable for mobile environments.

3. Computational model

The mobile networking environment consists of two distinct sets of entities: a set of MHs and a set of fixed hosts, i.e. MSSs. The set of MSSs and the communication channels among them form a static distributed system. On the other hand, the mobile processes can be viewed as a dynamic system. The MHs move in a geographical area, which is partitioned into cells. Each cell is covered by one MSS and MHs can only communicate with the MSS responsible for the cell in which it is located (and vice versa). An MH is connected to the system if and only if it is up and running and located in a cell covered by an alive MSS.

For the ease of the exposition, we assume the existence of a global discrete clock. This clock is a fictional device which is not known to the processes; it is only used to state specifications or prove protocol properties. The range *T* of clock values is the set of natural integers.

3.1 Mobile support stations: a static asynchronous distributed system

The set of MSSs and its underlying communication network is modeled as a static asynchronous system. The network of MSSs is made of a finite set of $n \geq 2$ fixed processes, namely, $B = \{b_1, \ldots, b_n\}$. Each MSS knows the identities of all MSSs. An MSS may correspond to a base station in the cellular network or a mesh router node in a wireless mesh network [5]. An MSS can fail by *crashing*, i.e., prematurely halting but it behaves correctly (i.e., according to its specification) until it possibly crashes. A process b_i is *correct* in a run of the leader election protocol if it does not crash in that run, otherwise it is faulty. We assume that a majority of MSSs are correct. In this paper, we use the following notations concerning the set *B* of MSSs:

- *t* denotes the maximum number of processes that can crash in a run ($1 \leq t < n/2$).
- $C \subseteq B$ is the set of MSSs that are correct in a run.

MSSs communicate by sending and receiving messages through reliable yet asynchronous channels. Each pair of MSSs $\{b_i, b_j\}$ is connected by a wired or wireless channel. Channels are reliable, i.e. they do not alter, create or lose messages. However, channels are asynchronous: the time to transfer a message from b_i to b_j is finite but unbounded.

Moreover, we consider that each MSS is provided with a query-response mechanism, which is the underlying communication approach used in our protocol. Such a query-response mechanism can be easily implemented in a time-free manner on top of a static asynchronous distributed system. More precisely, any MSS b_i can broadcast (to other MSSs) a *QUERY*() message and then wait for corresponding *RESPONSE*() from *n-t* MSSs (these are the *winning* responses for that query). The other *RESPONSE*() messages associated with the query, if any, are systemically discarded (they are the *losing* responses for that query).

A query issued by b_i is *terminated* if b_i has received *n-t* responses. We assume that a process issues a new query only when the previous one has terminated. Without loss of generality, the response from a process to its own query is assumed to always arrive among the first *n-t* responses it is waiting for. Moreover, *QUERY*() and *RESPONSE*() are assumed to be implicitly tagged in order not to confuse *RESPONSE*() messages corresponding to different *QUERY*() messages.

3.2. Mobile hosts: a dynamic system

Each mobile process has a unique identity. m_i denotes the MH whose identity is *i*. Like MSSs, MHs are asynchronous and suffer from crash failures. The system has infinitely many MHs $M = \{m_1, m_2, \ldots\}$. However, the join and leave of MHs satisfy the *finite arrival model* [2, 27], and each run of the protocol has only finitely many MHs. This means that there is no bound on the number of MHs for all runs but there is a bound on the number of MHs in each run. A protocol does not know that information because it varies from run to run.

An MH is connected only if it is located in a cell covered by some MSS that it is associated with. An MH can directly communicate with only the MSS located in its current cell. Messages between two MHs must be forwarded by corresponding MSSs. When an MH moves from one cell to another, a *handoff* procedure is then executed.

We use the following notation concerning the set M of MHs:

- $up(\tau) \subseteq M$ is the set of MHs that are connected to the system at time τ (i.e., $up(\tau)$ is the set of MHs that joined the system before time τ and no associated crash or disconnection occurs before time τ).

4. Problem definition and additional assumptions

Before we formally define the eventual leadership in the model described in system model described above, we first introduce the assumption on the stability of the system. The set of MHs is inherently dynamic. However, if each MH periodically join and then leave the system, being connected only for a short period (i.e., the system is unstable), it is not possible to elect an MH. Therefore, the system should exhibit stable periods that last long enough. The following set definition captures this notion of stability [18, 31]:

- $STABLE = \{m_i: \exists \tau \ s.t. \forall \tau' \geqslant \tau, \ m_i \ \square up(\tau')\}$. *STABLE* is the set of MHs that, once have entered the system, do not crash or get disconnected.

The set *STABLE* is the dynamic counterpart of the set C of correct processes defined in the context of static models. For our purpose, $STABLE \neq \varnothing$ is a necessary condition.

Then, the leadership of the leader oracle Ω can be defined as follows:

- *Eventual Leadership*: There is a time τ and a MH $m_l \in STABLE$ such that after τ, any invocation of *leader*() by any process m_i returns l.

As shown in [30], a leader oracle cannot be implemented in purely asynchronous systems where processes may crash. Therefore, some assumptions are needed to circumvent this impossibility. We suppose that each MSS b_i is able to gather partial knowledge about the presence of MHs in the system. We define the information available to MSSs in the failure detector framework [9].

We assume that each MSS b_i is equipped with a local failure detector that provides a set $local\_trust_i \subseteq M$, which provides hints on MHs that are currently up and connected. More precisely, at each MSS b_i, the set $local\_trust_i$ satisfies the following properties ($local\_trust^{\tau}_i$ denotes the value of $local\_trust_i$ at b_i at time τ):

- *Eventual Accuracy*: $\exists m \in STABLE; \exists \tau$ such that $\forall \tau' \geq \tau$: $m \in \cup_{i \in C} local\_trust^{\tau'}_i$.
- *Completeness*: If an MH m never joins the system, crashes or permanently leaves the system, then, $\exists \tau$ such that $\forall \tau' \geq \tau$: $m \notin \cup_{i \in C} local\_trust^{\tau'}_i$.

The accuracy property requires that eventually, at least one stable MH m is continuously trusted by the MSSs. Let us notice that it is not necessary that the same MSS permanently trusts m. On the contrary, we only require that after some time τ, $\forall \tau' \geq \tau$ there exists a correct MSS $b(\tau)$ that trusts m. Let $\tau < \tau_1 < \tau_2 < \tau_3 < ...$ be a sequence of time instants greater than τ. It is possible that $b(\tau_1) \neq b(\tau_2) \neq b(\tau_3), \ldots$ The completeness property requires that an MH that crashes or permanently leaves the system is eventually no longer trusted by any MSS.

Note that this failure detector does not provide much information on MHs present in the system. It only guarantees that eventually, at least one stable MH m is trusted by some MSS at each time instant, but it is possible that the *local_trust* sets permanently disagree. Since communications among MSSs are asynchronous and an MSS may have a different *local_trust* set at each time instant, MSSs cannot agree on the stable MH that they trust.

Moreover, our protocol depends on the following additional assumption, called MP_{dyn}. There is a stable MH m and a time τ (m and τ are not known in advance) such that at any time instant $\tau' \geq \tau$ there exists a set $Q^{\tau'} \subseteq B$ that satisfies the following property:

- $\forall \tau' \geq \tau$: $|Q^{\tau'}| \geq 2t + 1$.
- $\forall b \in Q^{\tau'}$: if b has not crashed by time τ', $m \in local\_trust^{\tau}_b$.

The assumption MP_{dyn} states that, eventually, there is a set of $2t + 1$ MSSs that trust the same MH. Moreover, this set can continuously change over time.

One concern is how to guarantee the assumption MP_{dyn}. One possible solution is to deploy a multiple coverage mobile network [1, 15, 16, 25]. Each point in the territory is covered by at least $2t + 1$ MSSs rather than only one MSS. Then, each MH keeps contact with at least $2t+1$ MSSs simultaneously.

5. A leadership facility for mobile networks

In this section, we first describe the operations of our protocol and then provide the correctness proof of the protocol.

5.1. Description of the protocol

The pseudocode of our protocol is shown in figures 1 and 2. "$\top$" is a special symbol that represents the whole universe of the mobile processes. Moreover, $\top \cap A = A$ (where A is any set of mobile processes). Our protocol extends the approach that previously appears in [11, 30, 31]. The MSSs act as servers to provide an eventual leadership service to the mobile processes. Each MSS b_i maintains a set $trust_i$, which consists of the MHs that are "globally" trusted by all MSSs in the view of b_i. Each $trust_i$ set is associated with a sequence number sn_i. sn_i is a logical date defining the "age" of $trust_i$.

```
//Code for MSS b_i
init: sn_i ←0; trust_i ←⊤
Task T1:
(01) repeat foreach j ∈ B do send PH1_QUERY() to b_j endfor;
(02)      wait until corresponding PH1_RESPONSE() has been
                    received from ≥n-t MSSs;
(03)      PH1_reci = {j : a PH1_RESPONSE() has been received
                    from b_j at line 03};
(04)      foreach j∈B do send PH2_QUERY(sn_i; trust_i) to b_j endfor;
(05)      wait until corresponding PH2_RESPONSE(LOCAL_TRUST)
                    has been received from ≥n-t MSSs;
(06)      PH2_reci = {j : a PH2_RESPONSE( ) has been received
                    from b_j at line 06};
(07)      let REC_FROM_i = ∪_{j∈PH1_reci∩PH2_reci} LOCAL_TRUST_j ;
(08)      trust_i ←trust_i∩REC_FROM_i
(09) end repeat

Task T2:
(10) upon reception of PH1_QUERY() from b_j:
(11)     query start_i[j] ←current_time(); send PH1_RESPONSE() to b_j
(12) upon reception of PH2 QUERY(sn_j ; trust_j) from b_j :
(13)     if sn_j = sn_i then trust_i← trust_i∩trust_j endif;
(14)     if sn_j> sn_i then trust_i ← trust_j ; sn_i ← sn_j endif;
(15)     if trust_i = Ø then trust_i ← ⊤; sn_i ← sn_i + 1 endif;
(16)     LOCAL_TRUST_i ←∪_{query_starti[j]≤current_time()} local_ trust^τ_i;
(17)     send PH2_RESPONSE(LOCAL_TRUST_i) to b_j
(18) upon reception of LEADER_QUERY(m) from mobile process m:
(19)     if trust_i = Ø∨ trust_i = ⊤ then l_i = m
(20)     else l_i = min(trust_i)
(21)      endif;
(22)      send LEADER(l_i) to m
```

Figure 1. Eventual leadership protocol: code for MSS

```
     When leader() is invoked:
(01)      send LEADER_QUERY(m) to current base station;
(02)      wait until LEADER(l) is received;
(03)      return (l)
```

Figure 2. Eventual leadership protocol: code for MH

When a mobile process m invokes the primitive *leader*(), it sends a *LEADER_REQUEST* message to its local MSS (Figure 2). On the reception of such a message, an MSS deterministically chooses an identity among the mobile processes it currently trusts and sends back this identity (lines 18-22).

The protocol consists of two tasks running in parallel at each MSS. Task T_1 is the core task in which each process initiates sequential queries and waits for corresponding responses. Task T_2 is triggered by the reception of messages. It implements the response mechanism associated with the queries: when a process b_i receives a query, it sends back a response carrying values with respect to the type of query received (lines 11 and 17).

An MSS b_i collects *local_trust* sets of other processes by sequentially issuing two-phase query-responses and then updates its $trust_i$ based on the responses. In the first phase, b_i broadcasts a *PH1_QUERY* and then waits for the response. When a process b_j receives such a query, it sends back a *PH1_RESPONSE* and starts recording the identities of processes that it locally trusts until it receives a *PH2_QUERY* from b_i. After b_i has collected response *PH1_RESPONSE* from at least n-t MSSs, it enters the second phase by broadcasting a *PH2_QUERY* with $< sn_i , trust_i >$ and then waits for response.

When an MSS b_j receives a pair $<sn_i, trust_i>$ (line 12), it updates $trust_j$ according to the respective values of sn_i and sn_j. If they are equal, it updates the set of trusted mobile processes to $trust_j \cap trust_i$ (line 13). If sn_i is greater, i.e., its current knowledge is too old, it adopts the set received (line 14). Otherwise, it discards the message received. If b_j then discovers that its set $trust_j$ is empty, b_j increases its sequence number sn_j and resets $trust_j$ to its initial value (line 15).

Then, b_j sends b_i a *PH2_RESPONSE* message, which carries the identities of the mobile processes that have been locally trusted by b_j since it has received the corresponding *PH1_QUERY* of b_i (lines 16-17).

After b_i has collected *PH2_RESPONSE* from at least n-t MSSs, it updates its $trust_i$ set based on the responses collected in the two phases, i.e.,

$$trust_i \leftarrow trust_i \cap (\cup_{j \ PH1\_reci \cap PH2\_reci} LOCAL\_TRUST_j).$$

After one or more query-response cycles, there eventually exists a finite age, after which the *sn* values no longer increase and the sets *trust* are (and remain) non-empty and equal. They actually converge towards to a subset of the *STABLE* set. The mobile process in these $trust_j$ sets with the smallest identity is then elected as the leader.

To see why this two-phase query-response cycle is necessary, let us assume that, if a process b_j received a query at a time τ, it sends back a response message that contains the value of $local\_trust_j$ at time τ. b_i collects *local_trust* sets of n-t MSSs, but these sets may have been "seen" at distinct times. Since the set Q of "witness" processes defined in property MP_{dyn} can change over time, it is possible that the *local_trust* sets collected by b_i does not satisfied any global property, even if the property MP_{dyn} is established. On the contrary, we will show in the proof that this two-phase query-response mechanism guarantees that the sets *REC_FROM* (i.e., the union of *local_trust* collected, line 07) eventually satisfy a global property. More precisely, Lemma 1 states that there exists a stable mobile process that eventually is always contained in any *REC_FROM* sets.

5.2. Correctness proof

In the following, x^{τ}_i denotes the value of the local variable x of process p_i (MH or MSS) at time τ. Given an execution, C is the set of MSSs that are correct in that execution. *STABLE* is the set of MHs that, after having entered the system, do not crash nor be disconnected.

Lemma 1. *There is a time τ and a stable MH m (i.e., m STABLE) such that every REC_FROM set computed (at line 07) after τ is such that m REC_FROM.*

Proof. Given an execution that satisfies the MP_{dyn} assumption, there is a time τ_0 and an MH m *STABLE* such that the following holds: $\tau \geq \tau_0$, there exists a set $Q^{\tau} \subseteq B$ such that (1) $|Q^{\tau}| \geq 2t+1$ and (2) $\forall b_i$ Q^{τ}: m $local\_trust^{\tau}_i$ or b_i has crashed before time τ.

Let us consider an MSS b_i that starts a query (at line 02) after τ_0. Let b_j be an MSS such that j $PH1\_rec_i \cap PH2\_rec_i$. This means that, for each phase of the query issued by b_i, the responses messages sent by b_j arrived among the first n-t ones at MSS b_i. Let τ_start_j be the time instant at which the *PH1_QUERY* from b_i is delivered to b_j and τ_end_j be the time at which b_j sends back the *PH2_RESPONSE* message. Note that the *PH2_RESPONSE* message sent by b_j carries the identities of all the MHs that has been trusted at least once by b_j during the time interval $[\tau\_start_j, \tau\_end_j]$ (lines 16-17).

The rest of the proof relies on the two following observations: *Observation O1*: $|PH1\_rec_i \cap PH2\_rec_i| \geq n\text{-}2t$ and *Observation O2*: $\exists \tau$ time instant such that: $\forall j$ $PH1\_rec_i \cap PH2\_rec_i$, τ $[\tau\_start_j, \tau\_end_j]$.

Let us consider the set REC_FROM_i computed by b_i after completing its query. This set is the union of the MHs that has been trusted by the MSSs b_j, j $PH1\_rec_i \cap PH2\_rec_i$ at some time instant between the beginning and the end of the two phase query of b_i. Let τ_1 be the time instant introduced in *O2*. In particular, $\cup_{j\ PH1\_rec_i \cap PH2\_rec_i} local\_trust_j^{\tau 1} \subseteq REC\_FROM_i$. As $\tau_1 \geq \tau_0$, it follows from the assumption MP_{dyn} that there exists at time τ_1 a set $Q^{\tau 1}$ of at least $\alpha \geq 2t + 1$ MSSs that either has crashed or trust m at time τ_1. Since $|PH1\_rec_i PH2\_rec_i| \geq n\text{-}2t$ (*O1*), it follows that $Q^{\tau 1} \cap (\cup_{j\ PH1\_rec_i \cap PH2\_rec_i} local\_trust_j^{\tau 1}) \neq \emptyset$, from which we conclude that m REC_FROM_i.

We have shown that there exists a time τ_i after which any REC_FROM_i set computed by b_i contains the identity of the stable mobile m. Taking $\tau_{max} = \max\{\tau_i: i \ B\}$ completes the proof.

Observation O1. For any MSS b_i that initiates and completes a two phases query, $|PH1\_rec_i \cap PH2\_rec_i| \geq n\text{-}2t \geq 1$.

Observation O2. Let b_i be an MSS that initiates and completes a two-phase query:

$\cap_{j\ PH1\_rec_i \cap PH2\_rec_i} [\tau\_start_j, \tau\_end_j] \neq \emptyset$

Due to the limit in space, we do not present the proof of the two observations in this paper.

□*Lemma* 1.

Lemma 2. *$\exists SN, \exists \tau$ such that, $\forall i$ B, $\forall \tau' \geq \tau$: i C ==> $sn_i(\tau') = SN$.*

Proof. Let τ_0 be a time such that (1) all faulty MSSs have crashed and (2) all messages sent by faulty MSSs have been delivered. Let τ_1 be the time defined in Lemma 1 and let $\tau_{clean} = \max(\tau_0, \tau_1)$. The idea is that after time τ_{clean} the system exhibits a "clean" behavior.

Let $SN^{\tau clean}$ be the maximal sequence number sn_i among the correct MSSs b_i at time τ_{clean}. Moreover, let say "the set *trusted* is associated with the sequence number *sn*" if there is a correct MSS b_j such that $trust_j = trusted$ and $sn_j = sn$ (let us observe that several sets can be associated with the same sequence number).

Claim C1. Let us assume that $\emptyset$ is associated with $SN^{\tau clean}$. There is then: (1) an MSS b_j that executes the reset statement at line 15, after which we have $(trust_j, sn_j) = (\top, SN^{\tau clean} + 1)$, and (2) the pair $(\top, SN^{\tau clean} + 1)$ is sent to all MSSs.

Due to the limit in space, we do not provide the proof of Claim *C1* in this paper.

We now show that $SN = SN^{\tau clean}$ or $SN = SN^{\tau clean} +1$. According to the definitions of τ_{clean} and $SN^{\tau clean}$, there exists a correct MSS b_i such that $sn_i = SN^{\tau clean}$. Due to the gossiping mechanism, after some time we will have $sn_j \geq SN^{\tau clean}$ for each b_j C. We consider two cases:

- *Case 1*: $\emptyset$ is never associated with $SN^{\tau clean}$. In that case, no correct MSS b_i will ever execute the reset statement at line 15. It follows that no MSS b_i will increase its sn_i variable, and the lemma follows.
- *Case 2*: $\emptyset$ is associated with $SN^{\tau clean}$. From Claim C2, there is an MSS b_j that eventually executes the reset statement at line 15, after which we have $(trust_j, sn_j) = (\top, SN^{\tau clean} +1)$, and this pair is sent to all the correct MSSs. This means that after some time, each MSS b_i will be such that $sn_i \geq SN^{\tau clean} +1$. As this occurs after time τ_1, the time defined in Lemma 1, it follows that, from now on, any set $trust_i$ permanently contains the stable MH m defined in Lemma 1. This is because each time b_i updates its set of trusted MHs (line 08), it intersects $trust_i$ which has been reset to $\top$ (the whole universe of MHs) with REC_FROM_i that always contains m. Consequently, no $PH2\_QUERY(\emptyset, SN^{\tau clean}+1)$ is

sent. Hence, no MSS can execute the reset statement at line 15, from which we conclude that no sequence number $>SN^{\tau clean}+1$ can be generated and the lemma follows.

□*Lemma* 2.

Theorem 1. *In any execution that satisfies the MP_{dyn} assumption, the protocol described in Figure 1 implements a leader facility in a mobile networking environment.*

Proof. Given a run that satisfies MP_{dyn}, let $PL = \cap\{trust_i\text{: } b_i \quad C \quad trust_i$ is associated with $SN\}$, where SN is defined in Lemma 2.

We first show that $PL \neq \emptyset$. Due to lemma 2, no sequence number greater than SN can be generated. This implies that $\emptyset$ cannot be associated with SN. Moreover, it follows from Lemma 1 and updates of $trust_i$ (line 08) that any $trust_i$ associated with SN contains the stable MH m introduced in Lemma 1.

We now show that $PL \subseteq STABLE$. This is a consequence of the completeness property satisfied by the $local\_trust_i$. More precisely, the completeness property states that an MH that crashes or gets disconnected from the system is eventually no longer locally trusted by each MSS b. Consequently, there is a time after which every *REC_FROM* does not contain crashed or disconnected MHs. Therefore, there is a time after which the REC_FROM_i contains only stable hosts. 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 MHs.

Finally, there is a time τ after which we have $\forall i$ C: $trust_i = PL$. This is a consequence of the finite arrival model (after some time, no more MH join the system) and the gossiping mechanism (lines 04 and reception of *PH2_QUERY* in task T_2). Let us consider an invocation made after τ of *leader*() that returns m_l. We have $m_l = \min(trust_i)$ where i is the identity of some correct MSS. Since $trust_i = PL \subseteq STABLE$, it follows that any of these invocations returns the same stable MH.

□*Theorem* 1.

6. Conclusions and future work

This paper investigates the eventual leader oracle Ω in the context of dynamic mobile networking environments, where MHs are associated with fixed mobile support stations to communicate with one another. MHs can join and leave the system at any time and the number of participating hosts can change arbitrarily as time passes. To implement Ω, we let MSSs act on behalf of MHs to elect an eventual unique MH as the leader. MSSs conduct the message exchange using a two-phase query-response mechanism in order to elect an eventual unique leader. Such a design can reduce the workload of MHs and the consumption of various resources, e.g. battery power and bandwidth. Since no assumption on the time property of processing speed or message delay, the proposed protocol is time-free and consequently especially attractive for mobile networks.

In future, we will evaluate the performance of our proposed protocol and compare it with similar work. Both numerical analysis and experimental simulations would be conducted.

Acknowledgements. This research is partially supported by Hong Kong University Research Grant Council under the CERG grant PolyU 5183/04E and France/Hong Kong Joint Research Scheme F-HK16/05T.

References

[1] Abbari B. J, Dinan E., and Fuhrmann W., Performance Analysis of a Multilink Packet Access for Next Generation Wireless Cellular Systems. *Proc. (PIMRC'98)*, 1998.

[2] Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004.

[3] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., On Implementing Omega with Weak Reliability and Synchrony Assumptions. *Proc. 22th ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 306-314, 2003.

[4] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 328-337, 2004.

[5] Akyildiz I. F., Wang X., and Wang W., Wireless mesh networks: a survey. *Computer Networks,* 47(4): 445-487, 2005.

[6] Anceaume E., Fernandez A., Mostefaoui A., Neiger G. and Raynal M., Necessary and Sufficient Condition for Transforming Limited Accuracy Failure Detectors. *Journal of Computer and System Sciences,* 68:123-133, 2004.

[7] Badache N., Hurfin M. and Macedo R. Solving the Consensus Problem in a Mobile Environment. *Proc. 18th IEEE Int'l Performance Computing and Communications Conference (IPCCC'99)*, IEEE, pp 29-35, Phoenix, AZ, 1999

[8] Badache N., Bouabdallah A. and Seba H., Solving the Consensus Problem in a Dynamic Group: an Approach Suitable for a Mobile Environment. *Proc. 7th IEEE Symposium on Computers and Communications (ISCC'02)*, IEEE, pp 29-35, Phoenix, AZ, 1999

[9] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

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

[11] Chu F., Reducing Ω to ◊*W. Information Processing Letters*, 76(6):293-298, 1998.

[12] Gupta I., Chandra T.D., and Goldszmidt G.S., On scalable and efficient distributed failure detectors. *Proc. 20th ACM symposium on Principles of distributed computing (PODC '01)*, ACM Press, pp. 170-179, 2001.
[13] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, 1988.
[14] Chockler G. and Malkhi D., Active Disk Paxos with Infinitely Many Processes. *Proc. 21th ACM symposium on Principles of distributed computing (PODC '02)*, ACM Press, pp. 78-87, 2002.
[15] Fernandes J., and Garcia J., Multiple coverage for MBS environments. *Proc. of PIMRC'00*, 2000.
[16] Fernandes J. and Garcia J., Cellular coverage for efficient transmission performance in MBs. *Proc. of VTC'00*, 2000.
[17] 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.
[18] Friedman R., Raynal M. and Travers C., Two Abstractions for Implementing Atomic Objects in Dynamic Systems. *Proc. 9th Int'l Conference on Principles of Distributed Systems (OPODIS'05)*, Springer Verlag, 2005.
[19] Friedman R. and Tcharny G., Evaluating Failure Detection in Mobile Ad-Hoc Networks. *Tech Report #CS-2003-06*, 22 pages, Computer Science Departement, The Technion (Israel) 2003.
[20] Gafni E. and Lamport L., Disk Paxos. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 330-344, 2000.
[21] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers,* 53(4):453-466, April 2004.
[22] Guerraoui R. and Raynal M., The Alpha and Omega of Asynchronous Consensus. *Tech Report #1676,* 21 pages, IRISA, University of Rennes 1 (Fance) 2005.
[23] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.
[24] Larrea M., Fern´andez A. and Ar`evalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symposium on Reliable Distributed Systems (SRDS'00)*, IEEE Computer Society Press, pp. 52-60, 2000.
[25] Lee W.C.Y., Overview of cellular CDMA. *IEEE Transactions on Vehicular Technology,* 40(2), May 1991.
[26] Malkhi D., Oprea F. and Zhou L., Ω Meets Paxos: Leader Election and Stability without Eventual Timely Links. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 199-213, 2005.
[27] 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.
[28] 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, 2003.
[29] Mostefaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.
[30] 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.
[31] Mostefaoui A., Raynal M., Travers C., Patterson S., Agrawal A. and El Abbasi A., From Static Distributed Systems to Dynamic Systems. *Proc. 24th Symposium on Reliable Distributed Systems (SRDS'05)*, IEEE Computer Society Press, pp. 109-118, 2005.
[32] Raynal M., A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. *ACM SIGACT News, Distributed Computing Column*, 36(1):53-70, 2005.