Designing Efficient Algorithms for the Eventually Perfect Failure Detector Class

Mikel Larrea, Alberto Lafuente, Iratxe Soraluze and Roberto Cortiñas
The University of the Basque Country, San Sebastián, Spain
Email: {mikel.larrea, alberto.lafuente, iratxe.soraluze, roberto.cortinas}@ehu.es

Joachim Wieland
RWTH Aachen, Aachen, Germany
Email: joachim.wieland@rwth-aachen.de

***Abstract*— This paper focuses on the design of unreliable failure detectors of the Eventually Perfect class ($\diamond\mathcal{P}$) in crash-prone partially synchronous systems. We adopt a monitoring mechanism based on heartbeats over a logical ring arrangement of processes as the common design feature. This provides good communication efficiency, a performance parameter which refers to the number of links that carry messages forever. We follow two different approaches that result in two families of failure detectors: a nearly communication-efficient family, which uses $n + \mathcal{C}$ links forever, being $\mathcal{C}$ the number of correct processes out of the n processes in the system, and a communication-efficient family, which uses only n links forever. Besides communication efficiency, we evaluate the algorithms in terms of QoS parameters, which include the capability of the failure detector to provide right answers as well as its reaction time.**

***Index Terms*— distributed algorithms, fault tolerance, Consensus, unreliable failure detectors**

I. Introduction

The concept of an *unreliable failure detector* was introduced by Chandra and Toueg [1] as a mechanism that provides (possibly incorrect) information about process failures. This mechanism has been used to solve several problems in asynchronous distributed systems, in particular the Consensus problem [2]. Chandra and Toueg characterized a class of failure detectors in terms of two properties: *completeness* and *accuracy*. Completeness characterizes the failure detector's capability of suspecting incorrect processes, while accuracy characterizes the failure detector's capability of not suspecting correct processes, i.e., restricts the mistakes that the failure detector can make. In this paper, we focus on the following completeness and accuracy properties: (i) *Weak Completeness:* eventually every process that crashes is permanently suspected by *some* correct process. (ii) *Strong Completeness:* eventually every process that crashes is permanently suspected by *every* correct process. (iii) *Eventual Strong Accuracy:* there is a time after which correct processes are not suspected by any correct process.

TABLE I.
TWO CLASSES OF FAILURE DETECTORS.

	Eventual Strong Accuracy
Strong Completeness	*Eventually Perfect ($\diamond\mathcal{P}$)*
Weak Completeness	*Eventually Quasi Perfect ($\diamond\mathcal{Q}$)*

Combining each one of the two completeness properties with the eventual strong accuracy property, we obtain two different classes of failure detectors, which are presented in Table I. Chandra and Toueg showed in [1] that any of the two classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$ can be used to solve the Consensus problem in an asynchronous system with a majority of correct processes. To do so, they first showed that classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$ are equivalent from a problem solvability point of view, i.e., a problem which is solvable with $\diamond\mathcal{P}$ is also solvable with $\diamond\mathcal{Q}$ and vice versa. It is worth noting that the equivalence of $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$ does not come for free, i.e., not all failure detectors in $\diamond\mathcal{Q}$ are in $\diamond\mathcal{P}$. Instead, it means that any failure detector in $\diamond\mathcal{Q}$ can be extended with a simple distributed algorithm to obtain a failure detector in $\diamond\mathcal{P}$.

Actually, Consensus can be solved with a weaker failure detector class called *Eventually Strong*, denoted $\diamond\mathcal{S}$, which satisfies strong completeness and eventual *weak* accuracy: there is a time after which *some* correct process is not suspected by any correct process. Note that any implementation of $\diamond\mathcal{P}$ trivially implements $\diamond\mathcal{S}$. Moreover, for certain problems [3] and Consensus protocols [4] failure detector $\diamond\mathcal{P}$ is required. Also, $\diamond\mathcal{P}$ is a more *natural* failure detector, in the sense that it ensures that eventually all correct processes output a list containing exactly the incorrect processes, providing a higher degree of accuracy. This may be a relevant quality of service (QoS) parameter for some applications.

Several algorithms implementing failure detector classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$ have been proposed in the literature. The algorithm proposed by Chandra and Toueg in [1] uses a heartbeat mechanism and all-to-all communication to detect faulty processes. The algorithms proposed by

Research partially supported by the Spanish Research Council, under grants TIN2006-15617-C03-01 and TIN2004-07474-C02-02, the Basque Government, under grant S-PE06IK01, and the Comunidad de Madrid, under grant S-0505/TIC/0285.

Aguilera et al. in [5] and by Larrea et al. in [6] use heartbeats too, and rely on a leader-based approach. On the other hand, the algorithms proposed by Larrea et al. in [7] use a polling —or query/reply— mechanism on a ring arrangement of processes. The leader-based and the ring-based algorithms are more efficient than the all-to-all algorithm regarding the number of messages sent (linear vs. quadratic). Observe also that compared to polling, the heartbeat mechanism reduces the number of messages to the half. Moreover, heartbeats inherently provide communication reliability in a system with fair lossy links, while polling usually requires reliable communication.

Recently, we have proposed several algorithms for implementing $\diamond\mathcal{P}$ that rely on a heartbeat-based detection mechanism over a ring arrangement of processes [8], [9]. Some of these algorithms outperform the former ones in terms of *communication efficiency*, a performance measure introduced in [5] consisting in the number of links that carry messages forever.

Our Contribution. In this paper, we revisit the design of heartbeat, ring-based algorithms for $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$ in order to implement efficient algorithms that provide good QoS. We start with a common skeleton defining the basic ring communication pattern. Then, we study the mechanisms required in order to provide the desired properties. We present two different approaches that result in two families of algorithms. The first family, which we denote as *nearly communication-efficient*, has been designed in a modular way. We start with a basic algorithm implementing $\diamond\mathcal{Q}$, and then a transformation is applied in order to get $\diamond\mathcal{P}$. The second family directly implements $\diamond\mathcal{P}$, and algorithms are communication-efficient, i.e. eventually just n links carry messages forever, being n the number of processes in the system. We have evaluated the performance of the algorithms in terms of QoS measures, mainly query accuracy probability and crash detection latency [10]. For that, we have developed and tested a number of $\diamond\mathcal{P}$ algorithms for each family. As we will see, some algorithms include sporadic extra communication, breaking the ring discipline. However, they always preserve the original communication efficiency characteristic of their family.

The rest of the paper is organized as follows. In Section II, we describe the system model considered in this work. In Section III, we discus the design of heartbeat, ring-based algorithms and describe the basic common skeleton to be used by all the algorithms. In Section IV, we describe the nearly communication-efficient family, while in Section V we present the communication-efficient family. In Section VI, we evaluate the performance of the algorithms. Finally, Section VII concludes the paper.

II. System Model

We consider a distributed system composed of a finite set Π of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate only by sending and receiving messages. Every pair of processes (p_i, p_j) is connected by two unidirectional and reliable communication links $p_i \rightarrow p_j$ and $p_j \rightarrow p_i$.

Processes can only fail by crashing, that is, by prematurely halting. Moreover, crashes are permanent, i.e., crashed processes do not recover. In every run of the system we identify two complementary subsets of Π: the subset of processes that do not fail, denoted $correct$, and the subset of processes that do fail, denoted $crashed$. We use $\mathcal{C}$ to denote the number of correct processes in the system in the run of interest, which we assume is at least one, i.e., $\mathcal{C} = |correct| \geq 1$.

We consider that processes are arranged in a logical ring. Without loss of generality, process p_i is preceded by process p_{i-1}, and followed by process p_{i+1}. As usual, p_1 follows p_n in the ring. In general, we will use the functions $pred(p)$ and $succ(p)$ respectively to denote the predecessor and the successor of a process p in the ring.

Concerning timing assumptions, we consider a partially synchronous model [1], [11] which stipulates that, in every run of the system, there are bounds on relative process speeds and on message transmission times, but these bounds are not known and they hold only after some unknown but finite time (called *GST* for *Global Stabilization Time*). Actually, the bounds must exist and hold only for the $\mathcal{C}$ links that eventually form the ring of correct processes, i.e., the links from every correct process to its correct successor in the ring. Hence, the bounds must only hold for a linear number of links.

Finally, in the algorithms presented in this paper we assume that a local clock that can measure real-time intervals is available to each process. Clocks are not synchronized.

III. Designing Ring-Based Failure Detector Algorithms

In this section, we describe a common core that will be used in all the algorithms presented in this work. The basic idea is to keep each process monitoring its correct predecessor in the ring by hearing heartbeats from it. Figure 1 presents the definitions and the three core tasks that try to converge, for every process p, to its correct predecessor in the ring. This core does not provide by itself the properties of any failure detector, and will be augmented with some mechanisms in order to every process p converge to both its correct predecessor and its correct successor in the ring (denoted $corr\_pred_p$ and $corr\_succ_p$ respectively), that will provide the properties of $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$.

In Figure 1, every process p starts sending periodically a heartbeat message to its successor in the ring, denoted by the variable $succ_p$ (Task 1). Also, every process p waits for periodical heartbeats from its predecessor in the ring, denoted by the variable $pred_p$. If p does not receive such a heartbeat on a specific time-out interval of $\Delta_p(pred_p)$, then p suspects that $pred_p$ has crashed, includes $pred_p$ in its local list of suspected processes, L_p, and sets $pred_p$ to $pred(pred_p)$ (Task 2). If later on p receives a heartbeat message from a process q it is erroneously suspecting,

Every process p executes the following:

```
( 1)  pred_p ← pred(p)        {p's estimated correct predecessor in the ring}
( 2)  succ_p ← succ(p)        {p's estimated correct successor in the ring}
( 3)  L_p ← ∅                 {L_p provides the properties of ◇Q}
( 4)  for all q ∈ Π:          {Δ_p(q) denotes the duration of p's time-out for q}
( 5)      Δ_p(q) ← default time-out interval

( 6)  cobegin

( 7)  || Task 1: repeat periodically
( 8)      if succ_p ≠ p then
( 9)          send (p-is-alive) to succ_p

(10)  || Task 2: repeat periodically
(11)      if pred_p ≠ p and p did not receive (pred_p-is-alive)
                during the last Δ_p(pred_p) ticks of p's clock then
(12)          L_p ← L_p ∪ {pred_p}      {p suspects pred_p has crashed}
(13)          pred_p ← pred(pred_p)

(14)  || Task 3: when receive (q-is-alive) for some q
(15)      if q ∈ L_p then               {p was erroneously suspecting q}
(16)          L_p ← L_p − {succ(pred_p), ..., q}
(17)          Δ_p(q) ← Δ_p(q) + 1
(18)          pred_p ← q

(19)  coend
```

Figure 1. Common core tasks for a heartbeat, ring-based failure detector.

p corrects L_p, increments the time-out interval $\Delta_p(q)$ in order to adapt the time-out, and changes the perception of its correct predecessor in the ring to q (Task 3).

Observe that the core itself does not modify the $succ$ variable of a process. We assume that there will be an out-of-the-core mechanism to change the $succ$ variable.

The following observation can be derived:

Observation 1: The list L_p of suspected processes of a process p contains exactly the processes between $pred_p$ and p, both excluded, i.e., when $pred_p \neq pred(p)$, $L_p = \{succ(pred_p), \ldots, pred(p)\}$, otherwise, when $pred_p = pred(p)$, $L_p = \emptyset$.

From now on, we will assume that any time instant t considered is larger than a time t_{base} that occurs after the stabilization time GST (i.e., $t_{base} > GST$), after every incorrect process has crashed, and after all messages sent by incorrect processes have been delivered. Note that this eventually happens. Hence, any new message delivered has necessarily been sent by a correct process. Then, we can enunciate the following lemma:

Lemma 1: For every correct process p, eventually and permanently $pred_p$ stabilizes on some correct process.

Proof: Note first that $pred_p$ is only modified in Task 2 (Line 13) and Task 3 (Line 18) of the algorithm. If $pred_p \notin correct$, since we are assuming that incorrect processes have already crashed and their messages have been delivered, by Task 2 p will suspect $pred_p$, include it in L_p, and set $pred_p$ to $pred(pred_p)$. Observe that this could happen even if $pred_p \in correct$, e.g., if $pred_p$ does not send any message to p or, even if doing so, if $\Delta_p(pred_p)$ is smaller than the bound on message transmission time. This scheme could be repeated at worst until p suspects every other process in the ring and sets $pred_p = p$ and $L_p = \Pi - \{p\}$. At any time, if p receives a message from a (necessarily correct) process $q \in L_p$, by Task 3 p removes q from L_p, increments the time-out $\Delta_p(q)$, and sets $pred_p$ to q. Hence, since we are assuming that incorrect processes have already crashed and their messages have been delivered, eventually either (1) p stabilizes $pred_p$ to some correct process $q \neq p$ because q sends a message periodically to p which is received before the time-out $\Delta_p(q)$ expires (e.g., because $\Delta_p(q)$ is bigger than the bound on message transmission time), or (2) the rest of processes are incorrect and p stabilizes $pred_p$ to p. ■

As we have said above, the core tasks do not solve by themselves some key questions required to provide the properties of the failure detectors we are interested in. First, it is not proved that $pred_p = corr\_pred_p$. Secondly, it is not defined how the $succ_p$ variable is updated, in order to converge to a correct process, specifically $corr\_succ_p$.

In order to complete the algorithm, two strategies can be followed. The first approach consists in introducing a specific interaction mechanism between p and $pred_p$, in order to ask $pred_p$ to update its successor to p. Doing like that, some messages will be sent in the opposite direction of heartbeats. The second approach consists in including into heartbeats global information about suspected process, and relying on the regular heartbeat communication to inform $pred_p$ about its successor. We present both approaches in the next sections.

IV. A Nearly Communication-Efficient Approach

In this section, we present a first approach to the design of algorithms implementing $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$. We introduce several additional tasks to allow a process p to update its successor $succ_p$. We start with a basic algorithm that implements $\diamond\mathcal{Q}$. Then, we transform the algorithm to implement a failure detector of the class $\diamond\mathcal{P}$, introducing some optimizations oriented to improve the QoS.

A. A Basic Ring-Based Algorithm for $\diamond\mathcal{Q}$

We present here a basic ring-based algorithm implementing $\diamond\mathcal{Q}$. For that, we add to the core skeleton of Figure 1 three new tasks. Specifically, every process p periodically sends a heartbeat message to the processes in the ring between itself and $succ_p$ (Task 4), and every process p periodically sends a ($START\_sending\_heartbeats$, p) message to its current predecessor in the ring $pred_p$ (Task 5). When a process p receives a ($START\_sending\_heartbeats$, new_succ) message, it sets $succ\_p$ to new_succ (Task 6).

Note that Task 1 and Task 4 could be integrated into a single task in which a process p periodically sends a heartbeat message to all processes in $\{succ(p), \ldots, succ_p\}$. Having two separated tasks, allows the use of independent periods for them, which is interesting from a performance point of view.

We show now that the algorithm of Figure 2 implements a failure detector of class $\diamond\mathcal{Q}$. The key of the proof is to show that eventually and permanently $pred_p = corr\_pred_p$ and $succ_p = corr\_succ_p$ for every correct process p. In other words, the ring stabilizes in terms of both the $pred$ and $succ$ variables of processes,

Every process p executes the following:

```
( 1)  pred_p ← pred(p)        {p's estimated correct predecessor in the ring}
( 2)  succ_p ← succ(p)        {p's estimated correct successor in the ring}
( 3)  L_p ← ∅                 {L_p provides the properties of ◇Q}
( 4)  for all q ∈ Π:          {Δ_p(q) denotes the duration of p's time-out for q}
( 5)      Δ_p(q) ← default time-out interval

( 6)  cobegin

( 7)  || Task 1: repeat periodically
( 8)      if succ_p ≠ p then
( 9)          send (p-is-alive) to succ_p

(10)  || Task 2: repeat periodically
(11)      if pred_p ≠ p and p did not receive (pred_p-is-alive)
                    during the last Δ_p(pred_p) ticks of p's clock then
(12)          L_p ← L_p ∪ {pred_p}        {p suspects pred_p has crashed}
(13)          pred_p ← pred(pred_p)

(14)  || Task 3: when receive (q-is-alive) for some q
(15)      if q ∈ L_p then                 {p was erroneously suspecting q}
(16)          L_p ← L_p − {succ(pred_p), ..., q}
(17)          Δ_p(q) ← Δ_p(q) + 1
(18)          pred_p ← q

(19)  || Task 4: repeat periodically
(20)      if succ_p ≠ succ(p) then
(21)          send (p-is-alive) to succ(p), ..., pred(succ_p)

(22)  || Task 5: repeat periodically
(23)      send (START_sending_heartbeats, p) to pred_p

(24)  || Task 6: when receive (START_sending_heartbeats,
                    new_succ)
(25)      succ_p ← new_succ

(26)  coend
```

Figure 2. Basic algorithm for $\diamond\mathcal{Q}$.

which guarantees the correct construction of the local lists L of suspected processes.

Lemma 2: For every correct process p, eventually and permanently $pred_p = corr\_pred_p$.

Proof: The proof is by contradiction. Since by Lemma 1 we have seen that eventually and permanently $pred_p$ stabilizes on some correct process, let us assume that eventually and permanently $pred_p \neq corr\_pred_p$. In the following, let be $q = pred_p$. Then, by Observation 1 $corr\_pred_p \in L_p$. Also, $succ_{corr\_pred_p} \in L_p$, i.e., $corr\_pred_p$ does not send messages periodically to p neither by Task 1 nor by Task 4, since otherwise p should set $pred_p$ to $corr\_pred_p$ (which is a contradiction). Finally, $succ_q$ must be set to p or further in the ring in order to p not suspecting q, and hence by Task 4 q sends messages periodically to $corr\_pred_p$. Clearly, eventually $pred_{corr\_pred_p} = q$, and by Tasks 5 and 6 of the algorithm eventually q will set $succ_q$ to $corr\_pred_p$, and hence q will stop sending messages periodically to p. Consequently, p will suspect q, which contradicts the fact that p stabilizes $pred_p$ to q. ■

Lemma 3: For every correct process p, eventually and permanently $succ_p = corr\_succ_p$.

Proof: Follows directly from Lemma 2 and Tasks 5 and 6 of the algorithm. ■

Theorem 1: The algorithm of Figure 2 implements a failure detector of class $\diamond\mathcal{Q}$.

Proof: From Lemmas 2 and 3, given a process p, if p is a correct process, then eventually p will be in the stable ring formed by correct processes. Otherwise, p is incorrect, and by Lemma 2 eventually and permanently $pred_{corr\_succ_p} = corr\_pred_p$. By Observation 1, p will eventually and permanently be included in $L_{corr\_succ_p}$. This provides the weak completeness property of $\diamond\mathcal{Q}$.

From the algorithm, p is never included in L_p. Once the ring has stabilized, by Observation 1 no correct process is included (in Task 2) in the lists L of any correct process. Hence, once the ring has stabilized no correct process will be present in any list of suspected processes. This provides the eventual strong accuracy property of $\diamond\mathcal{Q}$. ■

1) On a faster stabilization of the ring: We present here some modifications made to the basic algorithm of Figure 2 that lead to a faster stabilization of the ring, i.e., processes react faster to changes in the structure of the ring due to failures or erroneous suspicions. The modifications affect tasks 2, 3, 5 and 6 of the algorithm. Figure 3 presents the modified tasks in detail.

Every process p executes the following:

```
...

(10)  || Task 2: repeat periodically
(11)      if pred_p ≠ p and p did not receive (pred_p-is-alive)
                    during the last Δ_p(pred_p) ticks of p's clock then
(12)          L_p ← L_p ∪ {pred_p}        {p suspects pred_p has crashed}
(13)          pred_p ← pred(pred_p)
(a1)          if pred_p ≠ p then
(a2)              send (p, START_sending_heartbeats, p) to pred_p
(a3)          else   {p is suspecting the rest of processes: it sets succ_p to p}
(a4)              succ_p ← p

(14)  || Task 3: when receive (q-is-alive) for some q
(15)      if q ∈ L_p then                 {p was erroneously suspecting q}
(16)          L_p ← L_p − {succ(pred_p), ..., q}
(17)          Δ_p(q) ← Δ_p(q) + 1
(a5)          if pred_p ≠ p then
(a6)              send (p, START_sending_heartbeats, q) to pred_p
(a7)          else
(a8)              succ_p ← p's closest process following the ring,
                            either succ_p or q
(18)          pred_p ← q

...

(22)  || Task 5: repeat periodically
(a9)      if pred_p ≠ p then
(23)          send (p, START_sending_heartbeats, p) to pred_p

(24)  || Task 6: when receive (q, START_sending_heartbeats,
                    new_succ) for some q
(25)      succ_p ← new_succ
(a10)     if q ≠ new_succ then
(a11)         send (p-is-alive) to new_succ
```

Figure 3. Faster stabilization of the ring.

The first modification consists in sending a $(START\_sending\_heartbeats,\ p)$ message to $pred(pred_p)$ when p suspects $pred_p$ in Task 2 (Lines a1-a4), i.e., independently of the periodical activation of p's Task 5. However, we avoid that a process p sends $(START\_sending\_heartbeats,\ p)$ messages to itself. This occurs when p suspects the rest of processes in the system. In this case we have $pred_p = p$ after executing Line 13. Instead of sending the $(START\_sending\_heartbeats,\ p)$ message, p acts as if it instantaneously sent and received (in Task 6) the message: p directly sets its supposed correct successor in the ring to itself, i.e., $succ_p \leftarrow p$. With

this modification, we can avoid that a process p sends ($START\_sending\_heartbeats$, p) messages to itself in Task 5 (Line a9).

A second modification consists in sending a ($START\_sending\_heartbeats$, q) message to p's current $pred_p$ in Task 3 when p learns that it is erroneously suspecting q, and before setting $pred_p$ to q (Lines a5-a6). This helps in the stabilization of the ring, since by Task 6 p's current $pred_p$ will set its successor to q and thus redirect its heartbeats to q. As in the previous modification, in the case that p's current $pred_p$ was itself, i.e., $pred_p = p$, we could directly set $succ_p$ to q without sending any message. However, due to the reception of other ($START\_sending\_heartbeats$, -) messages it is possible that even having $pred_p = p$, p has $succ_p$ set to a process r other than p, in which case the best choice is to set $succ_p$ to p's closest process following the ring, either r (i.e., the current $succ_p$) or q (Lines a7-a8).

Finally, another modification consists in sending a heartbeat message to new_succ in Task 6 when the ($START\_sending\_heartbeats$, new_succ) message has been sent by Task 3 (Lines a10-a11). To distinguish these messages from those sent by tasks 2 and 5, we add the sender as a first parameter. Again, this speeds up the stabilization of the ring, since new_succ will receive as soon as possible a first heartbeat from p, independently of the periodical activation of p's Task 1.

Every process p executes the following:

```
( 1)   pred_p ← pred(p)        {p's estimated correct predecessor in the ring}
( 2)   succ_p ← succ(p)        {p's estimated correct successor in the ring}
( 3)   L_p ← ∅                 {L_p provides the properties of ◇Q}
(b1)   G_p ← ∅                 {G_p provides the properties of ◇P}
( 4)   for all q ∈ Π:          {Δ_p(q) denotes the duration of p's time-out for q}
( 5)       Δ_p(q) ← default time-out interval

...

(10)   || Task 2: repeat periodically
(11)       if pred_p ≠ p and p did not receive (pred_p-is-alive, −)
                       during the last Δ_p(pred_p) ticks of p's clock then
(12)           L_p ← L_p ∪ {pred_p}      {p suspects pred_p has crashed}
(b2)           G_p ← G_p ∪ {pred_p}
(13)           pred_p ← pred(pred_p)
(a1)           if pred_p ≠ p then
(a2)               send (p, START_sending_heartbeats, p) to pred_p
(a3)           else {p is suspecting the rest of processes: it sets succ_p to p}
(a4)               succ_p ← p

(14)   || Task 3: when receive (q-is-alive, G_q) for some q
(15)       if q ∈ L_p then                {p was erroneously suspecting q}
(16)           L_p ← L_p − {succ(pred_p), ..., q}
(17)           Δ_p(q) ← Δ_p(q) + 1
(a5)           if pred_p ≠ p then
(a6)               send (p, START_sending_heartbeats, q) to pred_p
(a7)           else
(a8)               succ_p ← p's closest process following the ring,
                                either succ_p or q
(18)           pred_p ← q
(b3)       if q = pred_p then
(b4)           G_p ← (G_q − {p}) ∪ L_p

...
```

Figure 4. Transformation of $\diamond\mathcal{Q}$ into $\diamond\mathcal{P}$ with linear detection latency.

B. Transforming $\diamond\mathcal{Q}$ into $\diamond\mathcal{P}$

A simple way for obtaining a $\diamond\mathcal{P}$ failure detector from $\diamond\mathcal{Q}$ is to apply the transformation algorithm proposed by Chandra and Toueg in [1], which has an all-to-all communication pattern. We rather want to keep the message complexity of the resulting algorithm linear, so we have first designed a transformation algorithm on top of the logical ring formed by processes. Consequently, this algorithm has a linear detection latency of real failures. Then, we present another transformation that has a lower, constant detection latency, based on the broadcast of suspicions.

1) Propagating Suspicions Through the Ring: Figure 4 presents a transformation of the $\diamond\mathcal{Q}$ algorithm of Figure 3 into $\diamond\mathcal{P}$. Besides its local list of suspected processes L_p, now every processes p has a global list G_p that provides the properties of $\diamond\mathcal{P}$. In order to get $\diamond\mathcal{P}$ out of $\diamond\mathcal{Q}$, processes propagate the global lists around the ring. Furthermore, instead of using specific messages to propagate the global lists, they are piggybacked in the heartbeat messages that processes send in the algorithm of Figure 3. This way there is no need of extra messages to implement $\diamond\mathcal{P}$. Every time a process p sends a heartbeat message either by Task 1, Task 4 or Task 6, it piggybacks its global list of suspected processes G_p in the heartbeat. Also, whenever p includes a process q in L_p (Task 2), p also includes q in G_p (Line b2). Finally, each time p receives a heartbeat message from its supposed correct predecessor ($pred_p$) in the ring (Task 3), p builds a new global list of suspected processes by merging the global list G_{pred_p} carried by the heartbeat (removing p if $p \in G_{pred_p}$) with its own local list L_p (Lines b3-b4).

We show now that the algorithm of Figure 4 implements a failure detector of class $\diamond\mathcal{P}$. First of all, it is easy to see that Observation 1, Lemma 1, Lemma 2, and Lemma 3 hold for this algorithm.

Observation 2: $L_p \subseteq G_p$ permanently for every process p.

Theorem 2: The algorithm of Figure 4 implements a failure detector of class $\diamond\mathcal{P}$.

Proof: From Lemmas 2 and 3, the ring formed exclusively by correct processes eventually stabilizes. Once this happens, every incorrect process q is permanently included in $L_{corr\_succ_p}$. By Observation 2, q is also permanently included in $G_{corr\_succ_p}$. By the algorithm, $G_{corr\_succ_p}$ is piggybacked into every heartbeat message sent by $corr\_succ_p$ in Task 1. By the periodical heartbeat messages sent by correct processes in Task 1 and Lines b3-b4 of the algorithm, eventually every correct process p will permanently include q in G_p. This provides the strong completeness property of $\diamond\mathcal{P}$. Finally, it is simple to see that the algorithm of Figure 4 preserves the strong accuracy property of $\diamond\mathcal{P}$ provided by the algorithm of Figure 2. ■

2) Broadcasting Suspicions to Reduce the Detection Latency: We present here another modification that, applied to the algorithm of Figure 4, allows to reduce the detection latency of real failures. The modification, presented in detail in Figure 5, affects Task 2 and introduces two additional tasks to handle two new types of messages

Every process p executes the following:

```
...
(10)   || Task 2: repeat periodically
(11)      if pred_p ≠ p and p did not receive (pred_p-is-alive, −)
                  during the last Δ_p(pred_p) ticks of p's clock then
(12)         L_p ← L_p ∪ {pred_p}          {p suspects pred_p has crashed}
(b2)         G_p ← G_p ∪ {pred_p}
(c1)         send (SUSPICION, p, pred_p) to the rest of processes
(13)         pred_p ← pred(pred_p)
(a1)         if pred_p ≠ p then
(a2)            send (p, START_sending_heartbeats, p) to pred_p
(a3)         else {p is suspecting the rest of processes: it sets succ_p to p}
(a4)            succ_p ← p
...
(c2)   || Task 7: when receive (SUSPICION, q, r) for some q
(c3)      if r ≠ p then
(c4)         G_p ← G_p ∪ {r} − {q}
(c5)      else          {p sends a refutation heartbeat to the rest of processes}
(c6)         send (REFUTATION, p) to the rest of processes
(c7)   || Task 8: when receive (REFUTATION, q) for some q
(c8)      G_p ← G_p − {q}
```

Figure 5. Transformation of $\diamond\mathcal{Q}$ into $\diamond\mathcal{P}$ with constant detection latency.

used for the notification of failure suspicions and refutations, respectively. Every time p suspects in Task 2 that $pred_p$ has crashed, p sends a ($SUSPICION$, p, $pred_p$) message to the rest of processes (Line c1), in order to let them know as soon as possible the (potential) crash of $pred_p$. In Task 7, when p receives a ($SUSPICION$, q, r) message for some q, if $r \neq p$ then p adds the suspected process r to its global list G_p, removing the sender of the message, i.e., q, from G_p. On the other hand, if $r = p$, i.e., p has been erroneously suspected by q, p sends a ($REFUTATION$, p) message to the rest of processes (Lines c2-c6). The reception of refutation messages is done in Task 8, consisting in the removal of the sender from G_p (Lines c7-c8). The modification does not affect the correctness of the algorithm, since after the stabilization of the ring no more ($SUSPICION$, -, -) messages will be sent, and hence Lines c2-c8 will never be executed.

V. A Communication-Efficient Approach

In this section, we present a second approach to the design of algorithms implementing $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$, that relies on including into heartbeats global information about suspected process. We start with a basic $\diamond\mathcal{P}$ algorithm, that is a minimal, non optimized version of the algorithm in [8]. As done in the first family of algorithms, several optimizations are also introduced in order to improve the QoS.

A. A Basic Communication-Efficient $\diamond\mathcal{P}$ Algorithm

Figure 6 presents the basic communication-efficient algorithm implementing $\diamond\mathcal{P}$. With respect to the core skeleton of Figure 1, in this algorithm every process p periodically sends a heartbeat message to the processes in the ring between itself and $succ_p$ (Task 4), as done in the algorithm of Figure 2. Also, besides its local list of suspected processes L_p, every process p has a global list G_p that provides the properties of $\diamond\mathcal{P}$. Processes propagate the global lists around the ring, piggybacked in the heartbeat messages they sent in Task 1 and Task 4. Whenever p includes a process q in L_p (Task 2), p also includes q in G_p. Finally, each time p receives a heartbeat message from its supposed correct predecessor ($pred_p$) in the ring (Task 3), p builds a new global list of suspected processes by merging the global list G_{pred_p} carried by the heartbeat (removing p if $p \in G_{pred_p}$) with its own local list L_p. Also, p sets its supposed correct successor ($succ_p$) in the ring to its nearest process following the ring not belonging to G_p.

Every process p executes the following:

```
( 1)   pred_p ← pred(p)      {p's estimated correct predecessor in the ring}
( 2)   succ_p ← succ(p)      {p's estimated correct successor in the ring}
( 3)   L_p ← ∅                          {L_p provides the properties of ◇Q}
( 4)   G_p ← ∅                          {G_p provides the properties of ◇P}
( 5)   for all q ∈ Π:     {Δ_p(q) denotes the duration of p's time-out for q}
( 6)      Δ_p(q) ← default time-out interval

( 7)   cobegin

( 8)   || Task 1: repeat periodically
( 9)      if succ_p ≠ p then
(10)         send (p-is-alive, G_p) to succ_p

(11)   || Task 2: repeat periodically
(12)      if pred_p ≠ p and p did not receive (pred_p-is-alive, −)
                  during the last Δ_p(pred_p) ticks of p's clock then
(13)         L_p ← L_p ∪ {pred_p}          {p suspects pred_p has crashed}
(14)         G_p ← G_p ∪ {pred_p}
(15)         pred_p ← pred(pred_p)

(16)   || Task 3: when receive (q-is-alive, G_q) for some q
(17)      if q ∈ L_p then                {p was erroneously suspecting q}
(18)         L_p ← L_p − {succ(pred_p), ..., q}
(19)         Δ_p(q) ← Δ_p(q) + 1
(20)         pred_p ← q
(21)      if q = pred_p then
(22)         G_p ← (G_q − {p}) ∪ L_p
(23)         succ_p ← p's nearest process following the ring ∉ G_p

(24)   || Task 4: repeat periodically
(25)      if succ_p ≠ succ(p) then
(26)         send (p-is-alive, G_p) to succ(p), ..., pred(succ_p)

(27)   coend
```

Figure 6. Basic communication-efficient implementation of $\diamond\mathcal{P}$.

Assuming a failure-free scenario, the number of messages periodically sent in the algorithm of Figure 6 in stability is n, since every process sends just one heartbeat to its successor in the ring. In the general case, the algorithm periodically sends n messages as well, due to the messages sent by Task 4 to crashed processes $\{succ(p), \ldots, pred(succ_p)\}$. As a consequence, the algorithm of Figure 6 is communication-efficient. In practice, and assuming that erroneous suspicions are not very frequent, the period of Task 4 could be bigger than the period of Task 1.

Some optimizations will be incrementally introduced in the algorithm of Figure 6 in order to provide a faster stabilization of the ring and reduce the detection latency of real failures, as we will see in the following subsections. It is important to note that communication efficiency is preserved in the new versions of the algorithm.

We show now that the algorithm of Figure 6 implements a failure detector of class $\diamond\mathcal{P}$. First of all, observe

that Observation 1 and Lemma 1 of Section III hold. We start by making an additional observation:

Observation 3: $L_p \subseteq G_p$ permanently for every process p.

The following two lemmas, identical to Lemma 2 and Lemma 3 of the previous section, require a new proof, since we do no longer rely on Tasks 5 and 6 of the algorithm of Figure 2.

Lemma 4: For every correct process p, eventually and permanently $pred_p = corr\_pred_p$.

Proof: The proof is by contradiction. Since by Lemma 1 we have seen that eventually and permanently $pred_p$ stabilizes on some correct process, let us assume that eventually and permanently $pred_p \neq corr\_pred_p$. Then, by Observation 1 $corr\_pred_p \in L_p$. Also, $succ_{corr\_pred_p} \in L_p$, i.e., $corr\_pred_p$ does not send messages periodically to p neither by Task 1 nor by Task 4, since otherwise p should set $pred_p$ to $corr\_pred_p$ (which is a contradiction). Finally, $succ_{pred_p}$ must be set to p at the closest in order to p not suspecting $pred_p$, and hence by Task 4 $pred_p$ sends messages periodically to $corr\_pred_p$. By Observations 1 and 3, eventually and permanently all incorrect processes $succ(corr\_pred_p), \ldots, pred(p)$ will be included in L_p, and hence transmitted in all the global lists G_p that p will send.

In general, we have for every correct process p that eventually and permanently (1) $succ_{pred_p}$ is set to p at the closest, and hence by Task 4 $pred_p$ sends messages periodically to $corr\_pred_p$, and by Task 4 or by Task 1 $pred_p$ sends messages periodically to p, and (2) $succ(corr\_pred_p), \ldots, pred(p) \in G_p$. A consequence of (1) is that eventually and permanently messages are propagated around a unique ring formed by all correct processes. This feature, combined with (2) and the fact that, by the algorithm, no other process in the ring removes any of the processes $succ(corr\_pred_p), \ldots, pred(p)$ included by p in the messages that are propagated, makes that eventually a list containing those processes will be received by $corr\_pred_p$. At the reception of that list, $corr\_pred_p$ will set $succ_{corr\_pred_p}$ to p at the closest (Line 23), and will start sending messages periodically to p. When p receives a message from $corr\_pred_p \in L_p$, by Task 3 p will set $pred_p$ to $corr\_pred_p$, which is a contradiction. ■

Lemma 5: For every correct process p, eventually and permanently $succ_p = corr\_succ_p$.

Proof: By Lemma 4, we have that for every correct process p, eventually and permanently $pred_p = corr\_pred_p$. Moreover, we have seen that eventually there is a unique ring formed by all correct processes around which the lists of suspected processes are propagated. Observe that, by the way global lists are constructed (Line 22), the list of suspected processes G_p propagated by p around the ring will permanently contain the incorrect processes in L_p, i.e., $succ(pred_p), \ldots, pred(p)$, and will never contain p. Also, no other process in the ring neither removes any of those processes nor adds p when constructing its own list of suspected processes. As a consequence, eventually $pred_p$ will permanently receive lists of suspected processes containing the processes in $succ(pred_p), \ldots, pred(p)$ and not containing p. Hence, by Line 23 of the algorithm $pred_p$ will permanently set $succ_{pred_p}$ to $p = corr\_succ_{pred_p}$, and the lemma holds. ■

Theorem 3: The algorithm of Figure 6 implements a failure detector of class $\diamond\mathcal{P}$.

Proof: From Lemmas 4 and 5, given a process p, if p is a correct process, then eventually p will be in the stable ring formed by correct processes, and will propagate by Task 1 its global list G_p, which is built based on the global list received from its correct predecessor in the ring $corr\_pred_p$. Otherwise, p is incorrect, and by Lemma 4 eventually and permanently $pred_{corr\_succ_p} = corr\_pred_p$. By Observations 1 and 3, p will eventually and permanently be included in $L_{corr\_succ_p}$ and in $G_{corr\_succ_p}$, which will be propagated around the ring. As a consequence, eventually and permanently p will be included in the global list of suspected processes of every correct process. This provides the strong completeness property of $\diamond\mathcal{P}$.

From the algorithm, every time a process p builds its global list of suspected processes G_p (in Task 3), p removes itself from G_p.[1] Once the ring has stabilized, by Observation 1 no correct process is included (in Task 2) in the lists L and G of any correct process. Hence, once the ring has stabilized and every correct process has built its global list of suspected processes in Task 3 (Line 22), no correct process will be present in any global list of suspected processes. This provides the eventual strong accuracy property of $\diamond\mathcal{P}$. ■

B. On a Faster Stabilization of the Ring

Figure 7 presents some modifications made to the basic algorithm of Figure 6 that lead to a faster stabilization of the ring, and a more accurate list G_p. The modifications affect tasks 2 and 3 of the algorithm, and introduce a new task.

A first modification consists in sending a ($START\_sending\_heartbeats$, p) message to $pred(pred_p)$ when p suspects $pred_p$ in Task 2 (Lines a1-a4). This new type of message will help processes to correct as soon as possible their $succ$ variables. Upon reception of a ($START\_sending\_heartbeats$, new_succ) message in the new Task 5, a process p sets $succ_p$ to new_succ (Lines a10-a11). However, we avoid that a process p sends a ($START\_sending\_heartbeats$, p) message to itself. This occurs when p suspects the rest of processes in the system. In this case we have $pred_p = p$ after executing Line 15 of the algorithm. Instead of sending the ($START\_sending\_heartbeats$, p) message, p directly sets $succ_p \leftarrow p$.

A second modification consists in sending a ($START\_sending\_heartbeats$, q) message to p's current $pred_p$ in Task 3 when p learns that it is

[1] Note that, from the algorithm, p is never included in L_p.

Every process p executes the following:

...

(11) || *Task 2:* **repeat periodically**
(12) **if** $pred_p \neq p$ **and** p did not receive ($pred_p$-is-alive, $-$) during the last $\Delta_p(pred_p)$ ticks of p's clock **then**
(13) $L_p \leftarrow L_p \cup \{pred_p\}$ {*p suspects $pred_p$ has crashed*}
(14) $G_p \leftarrow G_p \cup \{pred_p\}$
(15) $pred_p \leftarrow pred(pred_p)$
(a1) **if** $pred_p \neq p$ **then**
(a2) send ($START\_sending\_heartbeats, p$) to $pred_p$
(a3) **else** {*p is suspecting the rest of processes: it sets $succ_p$ to p*}
(a4) $succ_p \leftarrow p$
(16) || *Task 3:* **when** receive (q-is-alive, G_q) for some q
(17) **if** $q \in L_p$ **then** {*p was erroneously suspecting q*}
(18) $L_p \leftarrow L_p - \{succ(pred_p), \ldots, q\}$
(19) $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
(a5) **if** $pred_p \neq p$ **then**
(a6) send ($START\_sending\_heartbeats, q$) to $pred_p$
(20) $pred_p \leftarrow q$
(21) **if** $q = pred_p$ **then**
(22) $G_p \leftarrow (G_q - \{p\}) \cup L_p$
(23) $succ_p \leftarrow p$'s nearest process following the ring $\notin G_p$
(a7) **else** {*p receives a heartbeat from $q \notin L_p$ and $q \neq pred_p$*}
(a8) $G_p \leftarrow G_p - \{q\}$
(a9) send ($START\_sending\_heartbeats, pred_p$) to q

...

(a10) || *Task 5:* **when** receive ($START\_sending\_heartbeats, new\_succ$)
(a11) $succ_p \leftarrow new\_succ$
(a12) $G_p \leftarrow G_p - \{new\_succ\}$
(a13) send (p-is-alive, G_p) to new_succ

Figure 7. Faster stabilization of the ring. New code for tasks 2, 3, and new Task 5.

erroneously suspecting q, and before setting $pred_p$ to q (Lines a5-a6). This helps in the stabilization of the ring, since by Task 5 p's current $pred_p$ will set its successor to q and thus redirect its heartbeats to q.

A third modification consists in removing q from G_p and sending a message ($START\_sending\_heartbeats$, $pred_p$) to q in Task 3 when p receives a heartbeat message from $q \notin L_p$ and $q \neq pred_p$ (Lines a7-a9). Besides improving the accuracy of G_p, this helps in the stabilization of the ring as in the previous modification.

Finally, another modification consists in removing new_succ from G_p (to improve the accuracy of G_p) and sending a heartbeat message to new_succ in Task 5 (Lines a12-a13). This speeds up the stabilization of the ring, since new_succ will receive as soon as possible a new heartbeat from p, independently of the periodical activation of p's Task 1.

It should be pointed out that the $START\_sending\_heartbeats$ messages do not need to be reliably sent, i.e., they can be lost.

It is easy to see that the proposed modifications do not affect the correctness of the algorithm. Indeed, observe that the modifications do not affect the management of the $pred$ variables, which is the basis of the correctness proof of the basic algorithm of Figure 7. Also, note that the modified algorithm preserves that, eventually and permanently, the $succ$ variables are set to a correct process.[2]

[2]Actually, this is sufficient to implement $\diamond\mathcal{P}$. Nevertheless, to achieve communication efficiency the $succ$ variables must be set to the closest correct process following the ring.

The fact that eventually no ($START\_sending\_heartbeats$, $-$) message is sent indirectly shows that the algorithm of Figure 7 is communication-efficient as well.

C. Broadcasting Suspicions to Reduce the Detection Latency

Finally, the modification presented in Section IV-B.2 can also be applied to either the basic algorithm of Figure 6 or the improved one of Figure 7, reducing the detection latency of real failures. As previously, the modification does not affect neither the correctness of the algorithm nor its communication efficiency, since after the stabilization of the ring no more $SUSPICION$ messages will be sent.

VI. Performance Evaluation

In this section, we compare the performance of the $\diamond\mathcal{P}$ algorithms implemented over $\diamond\mathcal{Q}$ (described in Subsections IV-B.1 and IV-B.2, and henceforth referred to as LLW$_{QP1}$ and LLW$_{QP2}$ respectively), and the optimized versions of the communication-efficient $\diamond\mathcal{P}$ algorithms (described in Subsections V-B and V-C, and henceforth referred to as LLW$_1$ and LLW$_2$ respectively).

Besides results directly obtained from the analysis of the algorithms, we provide some QoS measures, obtained by simulation. Chandra-Toueg's all-to-all algorithm (CT) has been also evaluated as a reference.

A. Communication Efficiency

Assuming a failure-free scenario, the number of messages periodically sent in the algorithms LLW$_{QP1}$ and LLW$_{QP2}$ in stability is $2n$, since every process p sends one (p-is-alive) message to its successor in the ring (Task 1), and one ($START\_sending\_heartbeats$, p) message to its predecessor in the ring (Task 5). In the CT algorithm, this number is $n(n-1)$. Finally, the cost of the basic communication-efficient algorithm of Figure 6 (denoted LLW$_0$), as well as the LLW$_1$ and LLW$_2$ algorithms is n, since every process sends just one message to its successor in the ring.

TABLE II.
Performance analysis.

Algorithm	# messages (failure-free case)	# messages (general case)
CT [1]	$n(n-1)$	$\mathcal{C}(n-1)$
LLW$_{QP1}$ and LLW$_{QP2}$	$2n$	$n+\mathcal{C}$
LLW$_0$, LLW$_1$ and LLW$_2$	n	n

Table II summarizes the performance analysis of the different algorithms, including the general case with faulty processes (assuming that $\mathcal{C} \geq 2$ out of the n processes of the system are correct). In the general case, the algorithms LLW$_{QP1}$ and LLW$_{QP2}$ periodically send $n+\mathcal{C}$ messages, due to the messages sent by Task 1 ($\mathcal{C}$), Task 4 ($n-\mathcal{C}$) and Task 5 ($\mathcal{C}$). However, note that once all incorrect processes have crashed and the ring has stabilized, by Task 4

every correct process p sends heartbeat messages only to crashed processes $\{succ(p), \ldots, pred(succ_p)\}$. As a consequence, in practice, and assuming that erroneous suspicions are not very frequent, the period of Task 4 could be bigger than the period of Task 1. An adaptive alternative consists in setting initially the periods of Task 1 and Task 4 to similar values (to facilitate a fast reaction to erroneous suspicions during initial stabilization), and increasing the period of Task 4 every time it executes. This way eventually the number of messages sent by Task 4 will become negligible, and hence the number of messages periodically sent by this algorithm will tend to $2\mathcal{C}$. Furthermore, if the period of Task 5 is bigger than the period of Task 1, the number of messages periodically sent by this algorithm will tend to $\mathcal{C}$, which is optimal for $\mathcal{C}$ correct processes. Note that the same reasoning concerning the periodicity of Task 4 applies to the communication-efficient algorithms too.

Actually, the values of the general case of Table II correspond to the number of links that carry messages forever. Observe that the algorithms LLW_{QP1} and LLW_{QP2} are not communication-efficient, since they use more than n links. On the other hand, the algorithms LLW_0, LLW_1 and LLW_2 are communication-efficient.

B. Evaluating the Quality of Service

We present here performance results obtained by simulation. We have analyzed the following QoS measures:

- Query accuracy probability. It is the probability that a failure detection module that is queried by its associated process gives the right answer. This measure is based on [10], but has been enhanced in this work to apply to scenarios with more than just two processes,
- Crash detection latency. It is the time interval between the crash of a process and the time in which the rest of the processes suspect it in a permanent way. This measure quantifies how fast the failure detector reacts.

We have used the ns-2 simulator (http://www.isi.edu/nsnam/ns/) to compare the performance of the algorithms. In Table III we show the simulation settings for a typical local area network scenario. The simulation generates message delays at random with a uniform distribution. However, we have set minimum and maximum message bounds. Apparently, this contradicts our partially synchronous system model. Nevertheless, the algorithms do not exploit the knowledge of the maximal message delay when initializing the time-outs. This allows us to generate erroneous suspicions under the same conditions for different algorithms. Moreover, from a practical point of view the setting of a maximum message delay allows to determine the duration of the simulations.

The tests have been carried out for a number of nodes going from 3 to 24, using the settings of Table III. The query accuracy probability has been measured executing

TABLE III.
SIMULATION SETTINGS (IN SECONDS).

Parameter	Value
Minimum message delay	0.001
Maximum message delay	0.005
Periodicity of $ALIVE$ messages	0.5
Initial time-outs	0.5
Time-out increment	0.001

the algorithm during 2000 seconds, that has been empirically proved to be sufficient for comparative purposes. In fact, after this time the simulations have either stabilized or are near stabilization. We assume that no process crashes during the 2000 seconds. The crash detection time has been measured in a longer execution, introducing a crash at a time instant (2500 seconds) in which the system has stabilized. In both cases, every simulation has been repeated a sufficiently large number of times. In fact, the averages become stable after few executions.

Figures 8 and 9 summarize the average results obtained for each QoS measure. For clarity, instead of the probability of getting a right answer, we have used its complement, the bad answer probability, i.e., the probability that a failure detection module gives a wrong answer. Figure 8 shows that the bad answer probability is very similar and quite low for the algorithms of both families. For Chandra and Toueg's algorithm, the bad answer probability is negligible, at the price of using an all-to-all communication pattern periodically and forever. We can observe too that algorithms using a broadcast mechanism to notify suspicions have a higher bad answer probability. This is due to the fact that erroneous suspicions are notified to all the processes in the system, provoking that all the processes make a mistake for each erroneous suspicion.

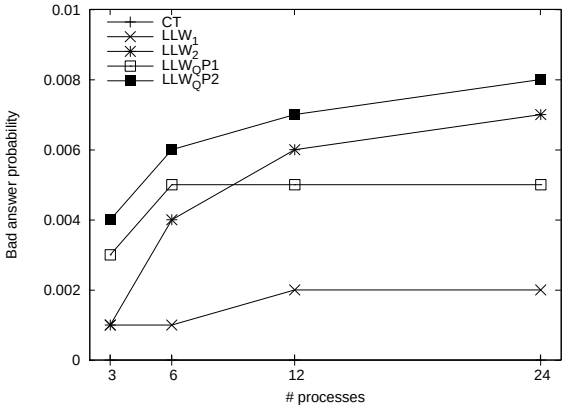

Figure 8. Bad answer probability.

This same strategy, i.e., the broadcast of suspicions, is very useful to reduce the crash detection time in the case of a real failure, as can be seen in Figure 9. For algorithms LLW_{QP1} and LLW_1, the crash detection time increases linearly with the number of processes; hence, these algorithms do not scale well for a large number of processes. On the other hand, for algorithms LLW_{QP2} and LLW_2, the crash detection time is constant and

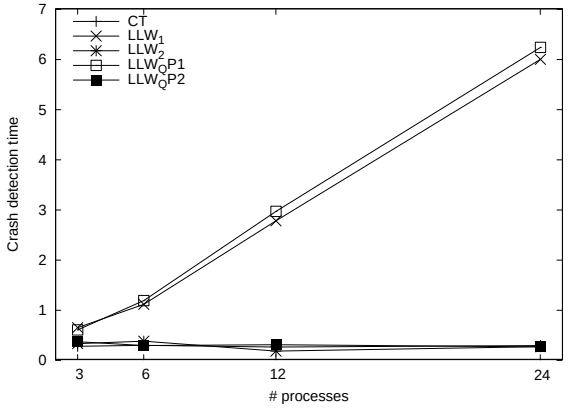

Figure 9. Crash detection time.

similar to Chandra and Toueg's algorithm, thanks to the broadcast of suspicions. It could be interesting to use this mechanism only once the system has stabilized and no more erroneous suspicions occur. We can consider that the systems is stabilizing when the number of failure suspicions drops below a given threshold. This strategy could help to provide a near optimal performance.

Finally, Figure 10 corroborates that, for the algorithms LLW$_{QP1}$ and LLW$_{QP2}$, setting the period of Task 5 higher than the period of Task 1 (4 and 16 times respectively), does not affect the query accuracy probability while reduces considerably the communication cost. For example, it passes from $O(2n)$ to $O(n)$ when the period of Task 5 is 16 times that of Task 1.

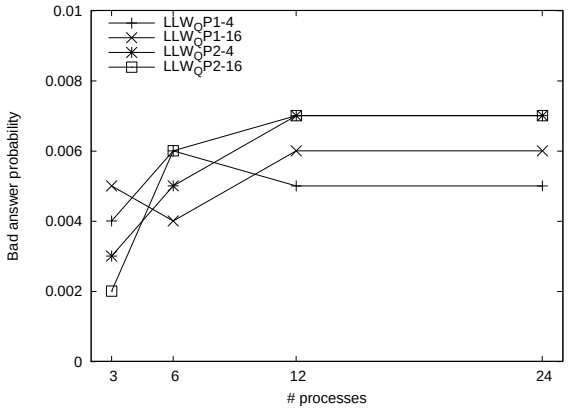

Figure 10. Incidence of task periodicity on bad answer probability.

VII. Conclusion

In this paper, we have explored the design of efficient failure detectors of the Eventually Perfect class ($\diamond\mathcal{P}$). We have identified core definitions and tasks shared by some heartbeat, ring-based algorithms we had developed previously. We have formally proved some basic properties of this core, and used it to derive two families of algorithms that follow two different approaches. The first family includes a set of algorithms that are nearly communication-efficient, and use $n + \mathcal{C}$ links forever. The second approach includes a minimal version of the communication-efficient algorithm of [8], and two variants that improve QoS parameters while preserving communication efficiency.

We have compared the algorithms in terms of two QoS parameters: query accuracy probability and crash detection time. From the simulation results, we have observed that both families exhibit similar QoS performance. Hence, we can conclude that the communication-efficient algorithms are a better choice, since they use a lower number of links. Also, we have observed that some variants perform better in the presence of real failures, while others do when erroneous suspicions occur. Interestingly, this fact can be exploited following a hybrid approach, in which the broadcast mechanism is disabled during stabilization, and later activated when the number of suspicions decreases, which indicates that the system is close to stabilization.

Acknowledgment

We are grateful to Neeraj Mittal for his helpful comments.

References

[1] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, March 1996.

[2] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, April 1980.

[3] R. Guerraoui, M. Kapalka, and P. Kouznetsov, "The weakest failure detector to boost obstruction-freedom," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC'2006)*. Stockholm, Sweden: LNCS 4167, Springer-Verlag, September 2006, pp. 399–412.

[4] W. Wu, J. Cao, J. Yang, and M. Raynal, "A hierarchical consensus protocol for mobile ad hoc networks," in *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'2006)*. Montbeliard-Sochaux, France: IEEE Computer Society, February 2006, pp. 64–72.

[5] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Stable leader election," in *Proceedings of the 15th International Symposium on Distributed Computing (DISC'2001)*. Lisbon, Portugal: LNCS 2180, Springer-Verlag, October 2001, pp. 108–122.

[6] M. Larrea, A. Fernández, and S. Arévalo, "Eventually consistent failure detectors," *Journal of Parallel and Distributed Computing*, vol. 65, no. 3, pp. 361–373, March 2005.

[7] M. Larrea, S. Arévalo, and A. Fernández, "Efficient algorithms to implement unreliable failure detectors in partially synchronous systems," in *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*. Bratislava: LNCS 1693, Springer-Verlag, September 1999, pp. 34–48.

[8] M. Larrea and A. Lafuente, "Brief announcement: Communication-efficient implementation of failure detector classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$," in *Proceedings of the 19th International Symposium on Distributed Computing (DISC'2005)*. Krakow, Poland: LNCS 3724, Springer-Verlag, September 2005, pp. 495–496.

[9] J. Wieland, M. Larrea, and A. Lafuente, "An evaluation of ring-based algorithms for the eventually perfect failure detector class," in *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2007*. Naples, Italy: IEEE Computer Society, February 2007, pp. 163–170.
[10] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 561–580, 2002.
[11] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, April 1988.

Mikel Larrea received his PhD degree in computer science from the University of the Basque Country in 2000, and his MS degree in computer science from the Swiss Federal Institute of Technology in 1995. He is currently an Assistant Professor of Computer Science at the University of the Basque Country. His research interests include distributed algorithms and systems, fault tolerance and ubiquitous computing.

Alberto Lafuente received his PhD degree in computer science from the University of the Basque Country in 1989, and his MS degree in computer science from the Technical University of Madrid in 1981. He is currently an Associate Professor of Computer Science at the University of the Basque Country. His research interests include distributed algorithms and systems, fault tolerance and ubiquitous computing.

Iratxe Soraluze received her PhD and MS degrees in computer science from the University of the Basque Country in 2004 and 1999 respectively. She is currently an Assistant Professor of Computer Science at the University of the Basque Country. Her research interests include distributed algorithms and systems, fault tolerance and ubiquitous computing.

Roberto Cortiñas is currently a PhD candidate at the University of the Basque Country. He received his MS degree in computer science from the University of the Basque Country in 1996. He is currently an Assistant Professor of Computer Science at the University of the Basque Country. His research interests include distributed algorithms and systems, fault tolerance and ubiquitous computing.

Joachim Wieland received his MS degree in computer science from the Aachen University in 2007. His research interests include distributed algorithms and systems, and fault tolerance.