Fault Tolerance Management in Distributed Systems: A New Leader-Based Consensus Algorithm

Fouad Hanna and Jean-Christophe Lapayre
Institute FEMTO-ST/DISC - UMR CNRS 6174
Franche-Comte University
Besancon, France
Email: (fouad.hanna, jc.lapayre)@femto-st.fr

Lionel Droz-Bartholet
Covalia Interactive
Temis Santé (Health Center)
Besancon, France
Email: lionel.droz@covalia.com

***Abstract*—It is well known that consensus algorithms are fundamental building blocks for fault tolerant distributed systems. In the literature of consensus, many algorithms have been proposed to solve this problem in different system models but few attempts have been made to analyze their performance. In this paper we present a new leader-based consensus algorithm (FLC algorithm) for the crash-stop failure model. Our algorithm uses the leader oracle Ω and adapts a decentralized communication pattern. In addition, we analyze and compare the performance of our algorithm to four of the most well-known consensus algorithms among asynchronous distributed systems of the crash-stop failure model. Our results give a global idea of the performance of these algorithms and show that our algorithm gives the best performance when process crashes take place in a system using a multicast network model. At the same time, our algorithm also gives a very acceptable performance, even when crashes occur in a unicast network model and in the case where no process crashes happen within the system.**

***Keywords*—fault tolerance; consensus; asynchronous distributed systems; unreliable failure detectors; leader oracle Ω**

I. INTRODUCTION

Consensus is a fundamental building block for our consistency protocol Ramos [1], a data consistency management protocol designed for and used in collaborative platforms (for example in medical tele-diagnosis: maximum number of participants 8). Our goal is to enhance the performance of our protocol and to develop a new version better suited to high process mobility environments. The choice of consensus algorithm affects the global performance of the protocol using it. In this paper, we study the performance of consensus by presenting a new leader-based consensus algorithm (the FLC algorithm) and comparing it to four of the most well-known algorithms of the crash-stop failure model. We take this model as a starting point because it is used by the current version of our consistency protocol Ramos [1]. We plan to study the performance of consensus algorithms in the crash-recovery failure model in a future investigation. In the crash-stop failure model, when a process crashes it stops participating in the algorithm and does not recover.

The new consensus algorithm that we present uses the leader oracle Ω to circumvent the impossibility result of the FLP theory. It adopts a decentralized communication pattern and tolerates a maximum number of process failures (process crashes) $f < \lceil \frac{n}{2} \rceil$. The principal idea of the algorithm is to use a simple leader election phase to reinforce the choice of the leader process made by the leader oracle Ω. At the same time, this leader election phase helps us to tolerate detection errors perhaps made by the leader oracle (in a case where several processes have different leaders proposed by the leader oracle). This leader election phase constitutes the first phase of the algorithm. During the second phase, each process sends the leader estimate to all others in order to make a decision on this value. We will refer to our algorithm as the FLC algorithm (**F***ouad H.*, **L***ionel D.*, **C***hristophe L.*) throughout the rest of the paper.

We analyze the performance of the FLC algorithm and compare it to four other well-known consensus algorithms, those of T. Chandra and S.Toueg [2], Paxos [3], A. Mostefaoui and M. Raynal [4] and also the latters leader-based algorithm [5]. These will be referred to as CT, Paxos, MR and MRLeader, respectively. These algorithms were chosen because they cover the four categories of consensus algorithm classification (Table I) and are the most popular in this domain. All these algorithms solve consensus in asynchronous distributed systems augmented with failure detectors [2]. They use different classes of failure detectors ($\diamond S$ and Ω) and different communication patterns (centralized and decentralized). Thus, comparison of these algorithms gives an overall idea of the influence of these elements on the performance of consensus algorithms and in particular shows how our own algorithm performs in comparison to them. We conducted our experiments by means of simulation using the Neko simulation and prototyping environment [6]. We considered two scenarios: the first is failure-free, where no process crashes, whereas in the second scenario multiple simultaneous processes crashes occur after system stabilization.

Other investigations have been undertaken to compare and analyze the performance of consensus algorithms. A. Coccoli et al. [7] analyze the performance of the CT consensus algorithm, particularly in cases of error by failure detectors. N. Sergent et al. [8] study the same algorithm to see how different implementations of failure detectors can impact its performance. L. Sampaio et al. [9] evaluate the influence of simultaneous round participation on the performance of

978-1-4799-5313-4/14/$31.00 ©2014 IEEE

consensus. P. Urban et al. [10] compare the performance of the CT algorithm with the MR algorithm. Similar work has been done by P. Urban et al. [11] to compare the CT algorithm with Paxos. Both of these papers ([10] and [11]) study the performance of consensus algorithms as part of an atomic broadcast algorithm and focus on the performance of the algorithms under different throughput values. The originality in our study is that we consider pure consensus runs and we compare the four algorithms simultaneously against our own to analyze the influence of all their different properties on their respective performances. In addition, we focus on examining the performance under a fixed throughput (5000 consensus runs) while varying the number of participants in the system.

In Section 2, we introduce the state of the art by defining the context of our work and introducing the existing consensus algorithms. In Section 3 we present our main contribution which consists of our new leader-based algorithm FLC and its performance study. The paper concludes in Section 4.

II. STATE OF THE ART

The context of our study consists of three subjects: the system model, the consensus problem and the unreliable failure detectors. We then give an overview and classification of the four well-known existing consensus algorithms that we use for comparison.

A. Context

Here we outline the system model used in this study. We also give some definitions concerning the consensus problem and the notion of unreliable failure detectors.

1) System Model: We consider a distributed system composed of a finite set of n processes $\{p_0, ..., p_{n-1}\}$. The system is asynchronous, which is to say that we do not make any time assumptions or place any bounds, neither on the message transmission delays nor on the execution speed of the processes. The communication network does not lose, alter or duplicate messages. We consider a crash-stop (fail-stop) model, which means that any process may crash with no possibility to recover (to come back into the system). A crashed process will perform no further operations.

2) The Consensus Problem: For a given set of processes, with their initially proposed values, the consensus problem is to select one of those values as the consensus or common value. The decision in this type of problem is irrevocable and the process cannot change its mind after deciding on a certain value. The following properties define the problem of consensus [2] [12] and they must therefore be satisfied by any consensus algorithm:

- ◇ *Termination*: every correct process will make a decision within a finite time.
- ◇ *Agreement*: all correct processes decide the same value.
- ◇ *Validity*: the decided value is one of the proposed values.

The solvability of the consensus problem was determined by the FLP theorem proven by M. Fischer, N. Lynch and M. Paterson [12] and which states that in an asynchronous distributed system, there is no algorithm that solves the consensus in every possible run in the presence of process failures [13]. The impossibility result of the FLP theorem comes from the fact that in asynchronous systems we cannot tell whether a process has really crashed or is simply running slowly or connected with a slow communication link. However, there are solutions to circumvent this impossibility result such as minimal synchronism [14], partial synchrony [15] and unreliable failure detectors [2]. In this study we examine the algorithms that use unreliable failure detectors.

3) Unreliable Failure Detectors: These were first introduced by T. Chandra and S. Toueg [2]. A failure detector is a module attached to each process and produces as output a list of processes that are currently suspected to have crashed. It therefore helps the processes to recognize crashed ones [16] [17]. Failure detectors are unreliable because they may make errors (wrong suspicions) by suspecting a correct process to have crashed or by not suspecting a crashed process. The behavior of failure detectors is characterized by two properties: *completeness* and *accuracy*. *Completeness* requires a failure detector to eventually suspect every process that has really crashed, while *accuracy* requires that a failure detector not suspect correct processes and therefore it restricts the errors that can be made by the failure detector. Several classes of failure detectors were defined based on different levels of these properties [2] [18]. We consider the class $\diamond S$ which is used by the algorithms CT [2] and MR [4]. For its *completeness* property, this class demands that eventually every crashed process be permanently suspected by every correct process. For its *accuracy* property, however, it demands that eventually some correct process never be suspected by any other correct process.
The leader oracle Ω is a special class of failure detectors [18] that is used by both Paxos [3] and MRLeader [5] algorithms. The output of this class of failure detectors is a single process currently considered to be correct. The leader oracle Ω satisfies the property of *eventual leadership*: There is a time after which all correct processes always trust the same correct process. For the leader oracle Ω, detection errors (wrong suspicions) can be seen as several processes with different leaders.

4) General Description and Definitions: The consensus algorithms considered in this study (including the new FLC algorithm) use failure detectors from classes $\diamond S$ and Ω to circumvent the impossibility of the FLP theory [12]. They are all round-based algorithms where execution proceeds in asynchronous rounds (i.e., not all processes necessarily execute the same round at a given time t). During each round, the processes attempt to make a decision. If a decision is made the execution ends. Otherwise, the processes proceed to the next round. Each round of the execution is managed by a process designated by the algorithm according to one of two approaches. The first is the rotating-coordinator approach in which the *coordinator* process is different for each round and is chosen by means of a mathematical rule. In this case we say that the algorithm is *coordinator-based*. The second approach is to use the leader oracle Ω and designate its output as the process that manages the round. We thus call this process the *leader* and the algorithm is said to be *leader-*

based. In this approach the *leader* process does not necessarily change for each execution round. The communication pattern between processes is said to be centralized if each process sends/receives messages only to/from the *coordinator* process (or the *leader*, if the algorithm is *leader-based*). In contrast, if all the processes communicate with each other, the communication pattern is said to be decentralized. All the algorithms tolerate a maximum number of failures (process crashes) $f < \lceil \frac{n}{2} \rceil$, i.e., there must be at least $\lceil (n+1)/2 \rceil$ correct processes in order to make a decision where n is the total number of processes in the system.

B. Existing Consensus Algorithms

In this subsection we briefly describe the four consensus algorithms that we consider for comparison, concentrating on their main characteristics. We then classify these algorithms based on their common points and differences.

1) CT Algorithm: This is a coordinator-based consensus algorithm presented by T. Chandra and S. Toueg in [2]. It uses the $\diamond S$ failure detector and a centralized communication pattern. The CT algorithm proceeds as follows:

- $\diamond$ First, each correct process sends its own estimate of the decision value to the *coordinator*.
- $\diamond$ The *coordinator* then collects $\lceil (n+1)/2 \rceil$ estimates in order to select the most recent and send it to all processes as their new estimate.
- $\diamond$ Next, each process not suspecting the *coordinator* to have crashed and having received his estimate, sends the *coordinator* an *ack* message. Otherwise, it sends a *nack* message.
- $\diamond$ Finally, the *coordinator* waits to receive $\lceil (n+1)/2 \rceil$ replies (*ack* or *nack*). If all the replies are *acks*, the *coordinator* then sends a decision message to all other processes.

2) Paxos Algorithm: The Paxos algorithm was presented by L. Lamport [3] [19]. It is a leader-based algorithm that uses the leader oracle Ω with a centralized communication pattern and proceeds as follows:

- $\diamond$ First, the *leader* sends a *prepare* message with the current round number to all other processes.
- $\diamond$ Upon receiving this message, a process responds with a *promise* message, if it has not yet accepted another *prepare* message with a higher round number. Otherwise, it responds with a *nack* message. The *promise* message also contains the current estimate of the process regarding the decision value.
- $\diamond$ Next, the *leader* waits for a majority of replies. If it receives at least one *nack* message it abandons the current round and proceeds to the following round. Otherwise, the *leader* sends an *accept* message containing the most recent estimate to all other processes. As before, each process responds with an *accepted* or a *nack* message. If all the replies are *accepted* messages, the *leader* sends a *decision* message to all other processes and decides on this estimate. Otherwise, it leaves the current round and proceeds to the next one.

3) MR Algorithm: This algorithm was presented by A. Mostefaoui and M. Raynal in [4]. It is a coordinator-based consensus algorithm that uses a $\diamond S$ failure detector with a decentralized communication pattern. The MR algorithm has a very simple structure in which the execution proceeds as follows:

- $\diamond$ First, the *coordinator* begins the round by diffusing its estimate of the decision value to all other processes.
- $\diamond$ Next, each process waits until it receives the *coordinator*'s estimate or until it suspects the *coordinator* to have crashed. If the estimate has been received, the process diffuses it to all other processes. Otherwise, if it suspects the *coordinator*, the process diffuses a null value $\perp$ to all other processes.
- $\diamond$ Upon reception of a majority of such messages, the process decides and diffuses a *decision* message if it has not received any null value. Otherwise, it proceeds to the next round with the estimate of the coordinator (if any has been received) as its local estimate of the decision value. At any time that a process receives a *decision* message, it decides on the value contained in the message if it has not yet decided and it diffuses the message to all other processes.

4) MRLeader Algorithm: The MRLeader algorithm was also presented by A. Mostefaoui and M. Raynal [5]. It is a leader-based algorithm that uses the leader oracle Ω but with a decentralized communication pattern. Each round of this algorithm proceeds as follows:

- $\diamond$ First, each process diffuses its local estimate to all other processes and waits to receive the *leader*'s estimate.
- $\diamond$ Next, each process diffuses the *leader*'s estimate that has been received and waits for a majority of messages carrying the same estimate.
- $\diamond$ If this is the case, the process then proceeds by again diffusing the estimate to all other processes to make sure that they have all adopted the same value. Otherwise, the process diffuses the null value.
- $\diamond$ Finally, each process waits for a majority of messages and if all of them contain the same value, the process decides and diffuses a *decision* message. If not, it proceeds to the next round.

5) Classification of the Algorithms: To facilitate the understanding of the algorithms and the differences in their performance, we have classified them according to the most important factors affecting their performance [10] [11] [7] [20]: The first factor is whether the algorithm is coordinator-based or leader-based and the second depends on its communication pattern. An outline is given in Table I.

TABLE I: Classification of the Consensus Algorithms

	Coordinator-based	**Leader-based**
Centralized	CT algorithm	Paxos algorithm
Decentralized	MR algorithm	MRLeader algorithm

III. CONTRIBUTION: A NEW CONSENSUS ALGORITHM

In this section we first present our contribution which consists of a new leader-based consensus algorithm (the FLC algorithm). Secondly, we introduce and analyze the results of a performance comparison obtained through a simulation using the Neko simulation and prototyping environment [6].

A. The new consensus algorithm (FLC)

The FLC algorithm is a leader-based consensus algorithm. It uses a decentralized communication pattern and assumes the existence of a majority of correct processes at any time (i.e., it tolerates a maximum number of process crashes $f < \lceil \frac{n}{2} \rceil$). Based on these characteristics, the FLC algorithm is in the same class as the MRLeader algorithm (Table I). We proved the correctness of the FLC algorithm by showing that it satisfies the properties of termination, agreement and validity (see [21] for details of proof of correctness).

The algorithm, described in Figure 2, is composed of three concurrent procedures. The $MainTask$ procedure represents the body of the algorithm. The $NewOmegaLeader(j)$ procedure is called by the leader oracle Ω when the leader changes, while the $DeliverMessage(msg)$ procedure is executed whenever a message is received. The principal idea of the algorithm is to ensure the existence of only one leader process per round by executing a simple election phase at the beginning of each round. Unlike other leader-based consensus algorithms (where a process becomes leader only based on the suggestion of its leader oracle), a process does not start a round r as leader unless it was elected by other processes to be leader of round r, i.e. received a majority of $LeaderAck(r)$ messages. This leader election phase reinforces the choice made by the leader oracle Ω, tolerates a minority of detection errors (even when the error is in the eventual leader process) and ensures the existence of only one leader process per round. Each process p_i manages a group of variables and exchanges two types of messages:

- $\diamond$ r_i contains the number of the current round, est_i contains the local estimate of the decision value.
- $\diamond$ $estFromLeader_i$ contains the estimate received from the leader process (if any is received) or the null value $\perp$.
- $\diamond$ $leaderChosen_i$ indicates when set to true that a non null estimate has been received from a leader process (in other words, a process has been elected to be leader and has proposed a value) or that the process p_i itself has been elected leader.
- $\diamond$ $voted_i$ indicates when set to true that process p_i has voted voted for a process to become leader (i.e. p_i has sent the $LeaderAck(r_i)$ message to a process).
- $\diamond$ $leaderChanged_i$ indicates when set to true that the process considered to be leader (based on the output of the leader oracle Ω) has changed after voting and before receiving a non null value from an elected leader process.
- $\diamond$ $nbLeaderVotes_i$ contains the number of votes that have been received to become the elected leader of the round (number of $LeaderAck(r_i)$ messages).
- $\diamond$ The counter variables $nbEST_i$, $nbNullEST_i$ and $nbNonNullEST_i$ contain, respectively, the total number of estimates (EST messages) (null or non null), the number of null estimates and the number of non null estimates received during round r_i.

The statement $exit(v)$ terminates the consensus execution (all three procedures) and returns the decision value (v) to process p_i. Each round of execution r_i is composed of two phases and proceeds as follows:

Phase 1 *(lines 7-10):* The aim of this phase is to elect a process to be leader of the round. Each process p_i sends a $LeaderAck(r_i)$ message to the process considered to be leader by its leader oracle Ω. A process p_j that receives a majority of these messages becomes leader of the round and executes line 31 allowing it to proceed to the second phase of the round (by setting $leaderChosen_j$ to true) and to diffuse its current estimate of the decision value to all other processes (by setting $estFromLeader_j$ to est_j). As mentioned above, only one process can be elected leader at the end of this phase and it will be the first one to break the waiting condition at line 10 with $estFromLeader_j \neq \perp$. Other processes wait (at line 10) for the reception of a non null estimate from the elected leader process (when $leaderChosen_i = true$). In general, we say that a leader is chosen for a process p_i (i.e. $leaderChosen_i = true$) if the process p_i is elected leader or when it receives a non null message from an elected leader process p_j where $j \neq i$. As the leader oracle Ω can make errors, we can have several processes voting for different leaders. If we have a majority of such errors then no process will receive a majority of votes and as a result no process will become elected leader. In this case, processes may bloc at line 10 and we will have two types of processes: the processes that voted for the eventual leader and those which made an error by voting for another process. For the latter, they will eventually have their considered leader process changed $leaderChanged_i = true$ and will proceed by sending a null estimate to all other processes. This is because the leader oracle Ω will eventually detect the error and select the correct process to be considered leader (by calling the $NewOmegaLeader(j)$ procedure). So as to prevent this blocking, process p_i waits also at line 10, either to receive a null estimate or to have $leaderChanged_i = true$. At the end of this phase, each process p_i that breaks the waiting condition at line 10 will have $estFromLeader_i = v$ or $\perp$ where v is the non null value that was sent by the elected leader process.

Phase 2 *(lines 12-18):* The aim of this phase is to allow a process to decide without compromising the agreement property of the algorithm. In order to attain this goal, each process p_i diffuses its $estFromLeader_i$ variable (by sending the EST message at line 12). As a result of the first phase, each process will diffuse v which is the value proposed by the elected leader or the null value $\perp$. Each Process p_i waits to receive a majority of EST messages containing either v or the null value $\perp$. If process p_i receives a majority of non null messages ($nbNonNullEST_i \geq \lceil \frac{n+1}{2} \rceil$) then it decides on the value v and sends a decision message $Decide(v)$ to all other processes and terminates the execution. If it receives a majority of the null value $\perp$ ($nbNullEST_i \geq \lceil \frac{n+1}{2} \rceil$) it then

skips to the next round. If it has neither a majority of v nor of the null value $\perp$, it means that the process received at least one non null value v and it proceeds to the next round by executing line 17 and setting its local estimate to the value v. This guarantees that if a process p_i decides v during round r and if another process p_x proceeds to (r+1) then p_x does it with $est_x = v$.

The $NewOmegaLeader(j)$ procedure is called by the leader oracle Ω to inform the process that the process considered to be leader has changed. This change is taken into account only if the process has not yet finished the first phase, or, more precisely, if the process has voted but not yet chosen a leader. If this is the case, the process proceeds to phase 2 by sending a null estimate $\perp$ to all other processes. To elect a leader a process votes only once during a round and when a change of leader takes place it does not vote for the new considered leader during the same round.

The $DeliverMessage(msg)$ procedure is called when a message is received during round r_i. The behavior of this procedure is determined according to the type of received message. If it is a $LeaderAck(r_i)$ then the number of votes to be leader $nbLeaderVotes_i$ will be increased by one only if the process has not yet chosen a leader ($leaderChosen_i = false$). The process becomes the elected leader of the round when $nbLeaderVotes_i \geq \lceil \frac{n+1}{2} \rceil$. If the message is a decision message, the process also decides the same value, sends a decision message to all other processes and terminates the execution. If the message is an estimate message EST, then it will increase the counters according to the received value (null or non null); if a non null value is received before having chosen a leader, then the process executes line 31 to take into account the value sent by the elected leader.

Figure 1 illustrates execution of the FLC algorithm. In this execution there are neither process crashes nor detection errors. We have three processes and all the leader oracles Ω indicate p_0 as the considered leader.

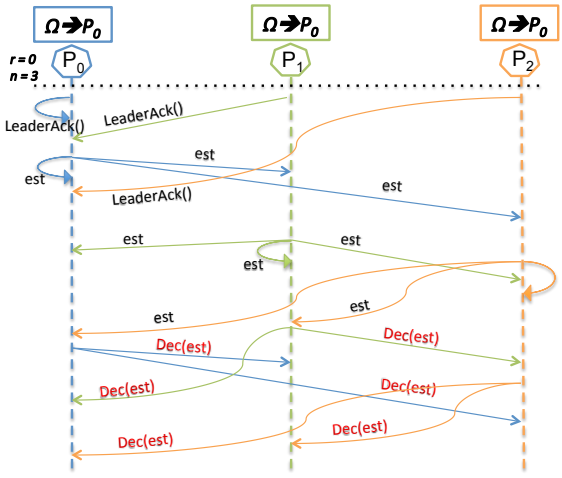


Figure 1: FLC algorithm - execution scheme

B. Experiments and Performance Results

In this subsection we present the results of our experiments. We start by introducing the simulation platform Neko. We next explain the different scenarios and parameters that we have tested in this study and we give the most significant results obtained.

1) Neko Platform and Simulation Models: We conducted our experiments via simulation and implemented the five consensus algorithms on top of the Neko simulation and prototyping environment [6]. Neko is written in Java and it allows simulation and execution of distributed algorithms.

Figure 3 shows our simulation model and illustrates how an experiment is conducted. Each process is composed of several modules that are used either for executing the consensus algorithm or for controlling the execution of an experiment. The module *Consensus* contains the implementation of the consensus algorithm that we want to execute. In order to obtain accurate measurements, i.e. all processes start the consensus algorithm at the same time, the module *clock* is used to synchronize the local clock of the processes. Only one process (i.e. p_0 in Figure 3) contains the module *Test Coordinator* which is responsible for starting an execution, and for gathering and analyzing the measurements sent by processes at the end of the execution. The *Test Client* module exists in every process and is responsible for starting the consensus algorithm and measuring its execution time in order to send it to the *Test Coordinator* module. To simulate the communication network we used the same model as in [11] [10]. This model accounts for resource contention of the network by providing a parameter λ to control it. λ shows the relative speed of processing a message on a host compared to transmitting it over the network. By changing the value of λ we can represent different network environments. We conducted our experiments with the frequently used values of λ : 0.1, 10, 1 which represent, respectively, a WAN, LAN and an intermediate network configuration [11] [10] [9]. For lack of space, we show here only the results obtained when λ = 1 (refer to [22] for all results). In our experiments we also used two variations of the network model: one supports multicast messages while the other supports only unicast messages (point-to-point model).

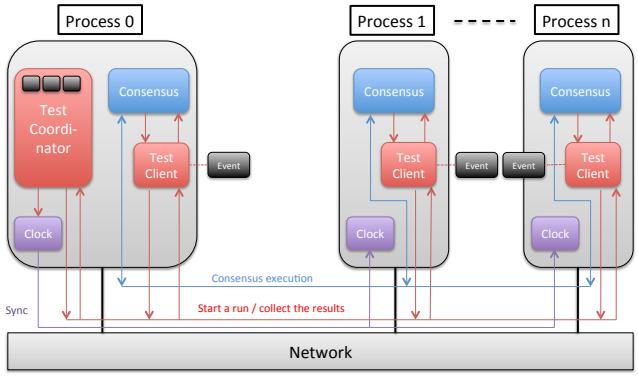


Figure 3: Simulation Model - Running an Experiment

Performance Metric: Our performance metric is the mean execution time defined by the time elapsed between the beginning and the end of the execution of the consensus algorithm (the end of the execution is reached when a process decides). For every run of the consensus algorithm we took the average execution time of all the processes to obtain the

```
1: procedure MAINTASK                                                    ▷ The body of the algorithm
2:     r_i ← −1; est_i ← v_i;                                                          ▷ v_i ≠⊥
3:     while true do
4:         r_i ← r_i + 1; estFromLeader_i ←⊥; leaderChosen_i ← false;
5:         voted_i ← false; leaderChanged_i ← false; nbLeaderVotes_i ← 0;
6:         nbEST_i ← 0; nbNullEST_i ← 0; nbNonNullEST_i ← 0;
7:         leader_i ← getOmegaLeader();
8:         send LeaderAck(r_i) to leader_i;
9:         voted_i ← true;
10:        wait until (leaderChosen_i = true ⋁ leaderChanged_i = true ⋁ nbNullEST_i > 0);
11:                                          ▷ At this point we have estFromLeader_i = v or ⊥
12:        ∀j: send EST(r_i, leader_i, estFromLeader_i) to j;
13:        wait until (nbEST_i ≥ ⌈(n+1)/2⌉);                ▷ Wait to receive a majority of EST messages
14:        if nbNullEST_i ≥ ⌈(n+1)/2⌉ then skip;                                  ▷ Skip to the next round
15:        else if nbNonNullEST_i ≥ ⌈(n+1)/2⌉ then
16:            ∀j ≠ i: send Decide(v) to j; exit (v);                ▷ Terminate the consensus execution
17:        else est_i ← v;
18:        end if
19:    end while
20: end procedure

21: procedure NEWOMEGALEADER(j)                          ▷ Called by Ω when the leader changes
22:    if voted_i = true ∧ leaderChosen_i = false then
23:        leader_i ← j; leaderChanged_i ← true;
24:    end if
25: end procedure

26: procedure DELIVERMESSAGE(msg)                                ▷ Called when a message is received
27:    if msg = LeaderAck(r_i) then
28:        if leaderChosen_i = false then
29:            nbLeaderVotes_i + +;
30:            if nbLeaderVotes_i ≥ ⌈(n+1)/2⌉ then                          ▷ Process i is elected leader
31:                estFromLeader_i ← est_i; leader_i ← i; leaderChosen_i ← true;
32:            end if
33:        end if
34:    else if msg = Decide(v) then ∀j ≠ i: send Decide(v) to j; exit (v);   ▷ Terminate the consensus execution
35:    else if msg = EST(r_i, leader, v) then
36:        if v ≠⊥ then
37:            nbNonNullEST_i + +;
38:            if leaderChosen_i = false then
39:                estFromLeader_i ← v; leader_i ← leader; leaderChosen_i ← true;
40:            end if
41:        else nbNullEST_i + +;
42:        end if
43:        nbEST_i + +;
44:    end if
45: end procedure
```

Figure 2: FLC consensus algorithm

algorithms mean execution time. As in other performance studies [11] [10] [9], measurements were obtained by running the same consensus algorithm many times (5000 runs in our study) and averaging the mean execution times obtained for each run. Execution time is measured in milliseconds *ms*.

Modeling Failure Detectors: We used an abstract model of failure detectors based on their quality of service parameters [23]. The only parameter that is relevant to our execution scenarios is the detection time DT: the time lapse between the crash of a process and when the monitoring process begins to suspect it permanently. The leader oracle Ω is modeled on top of the $\diamond S$ failure detector such that its output is the first correct process (ordered by their index) among the set of processes that are currently considered correct [24]. In our experiments detection time DT is the same for all processes. Due to the short execution time of consensus algorithms, failure detectors are not reset to their initial state at the beginning of each consensus run [7].

2) Scenarios and Results: In our study we considered two scenarios for the execution of consensus algorithms. The

first contains neither process crashes nor detection errors by failure detectors, whereas in the second scenario we injected multiple simultaneous crashes into the execution. For each scenario we conducted experiments on the different values of network contention parameter λ (0.1, 10, 1) and for the two network models (multicast and unicast). For each configuration we varied the number of processes *n* from 3 to 10. We do not investigate large numbers of processes (scalability) because our work is oriented towards collaborative distributed applications where the number of processes (practitioners for telediagnosis for example) is never more than 8 and therefor we conducted our experiments with up to 10 processes. In order to obtain the mean execution time for a certain number of processes, we executed a series of 5000 consensus runs. All the graphs show mean execution time vs. number of processes.

Scenario 1*:* This scenario represents the best conditions for consensus execution. None of the processes crashed during the experiment and failure detectors did not make errors (no wrong suspicions). Figure 4 shows the results obtained for this scenario in a multicast network (Figure 4a) and in a unicast network (Figure 4b).

We observed that the FLC algorithm gave a remarkably better performance than the MRLeader algorithm in the multicast and the unicast network model despite the fact that the two algorithms share the same classification (both are decentralized and leader-based). The MR algorithm performed better than all other algorithms in both the multicast and unicast network models despite its decentralized communication pattern. MRLeader, CT and Paxos algorithms approximately gave the same performance in the multicast network model. However, when it comes to the unicast network model, we observed a broad degradation in performance for the MRLeader algorithm.

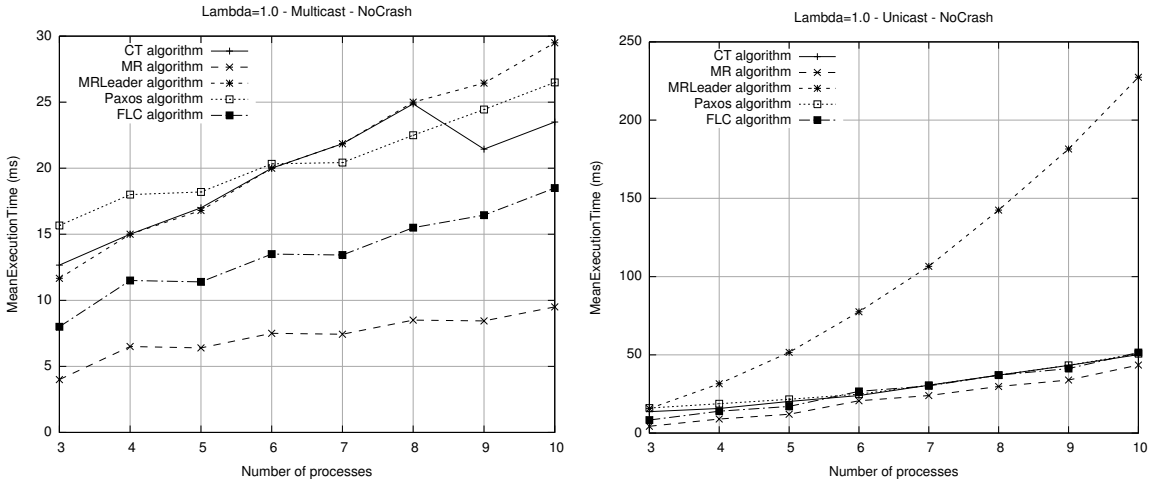
Scenario 2*:* In this scenario we examine the influence of process crashes on algorithm performance. We injected multiple simultaneous process crashes after the stabilization of the system, i.e. the system starts without crashes, but they occur after a predefined number of consensus runs. Failure detectors do not make errors. In order to compare the algorithms in the worst case scenario, the crashed processes are coordinator/leader processes. In our experiments we considered that, for a coordinator-based algorithm, process p_i is the coordinator of round r_i. For a leader-based algorithm the leader process is the correct process with the smallest index (we start with p_0 and if it crashes then the leader is p_1, etc.). Figure 5 shows the results obtained with a multicast network model in the case of one crash (Figure 5a), two crashes (Figure 5b) and three crashes (Figure 5c). Figure 6 shows the same results, but with a unicast network model (Figure 6a: one crash, Figure 6b: two crashes and Figure 6c: three crashes).

For the multicast network model we observe that the CT algorithm gives the worst performance in all three cases. With the number of system crashes, the performance of the MR algorithm gradually deteriorates (compared to other algorithms). On the other hand, the performance of the leader-based algorithms (MRLeader, Paxos and FLC) gradually improves (compared to other algorithms) with number of crashes. The FLC algorithm performs the best in all three cases. When we use a unicast network model, the results change: the performance of the centralized algorithms (CT and Paxos) is better than the two decentralized algorithms MR and MRLeader in all the three cases. The Paxos algorithm (which is both decentralized and leader-based) gives the best performance. We observe that our algorithm (FLC) gives a remarkably better performance than the other decentralized algorithms (MR and MRLeader) and an even better one than the CT algorithm which uses a centralized communication pattern. After three simultaneous crashes MRLeader is better than the MR algorithm.

3) Discussion: Here we discuss the performance results previously obtained based on the design and the properties of each consensus algorithm.

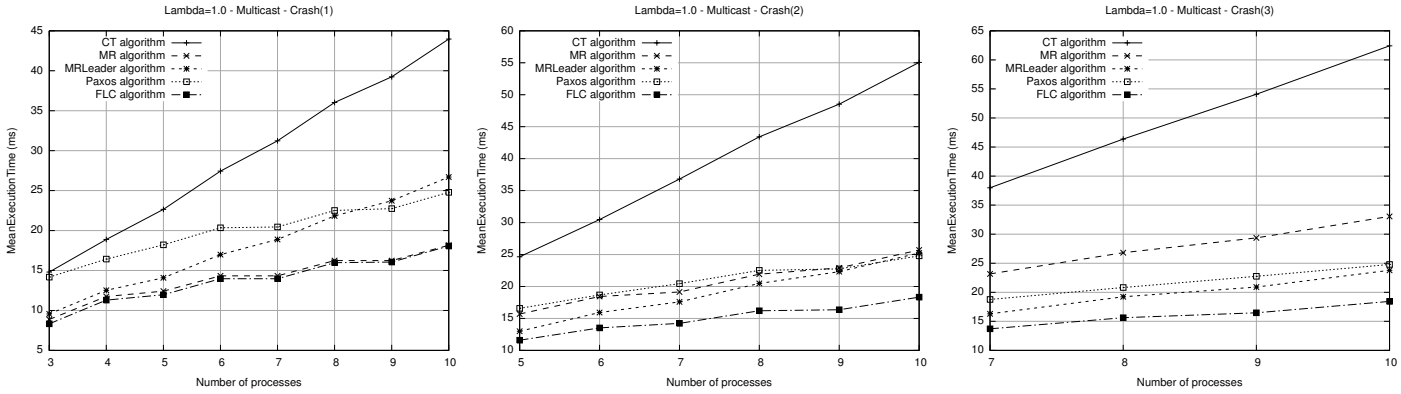
We start with Scenario 1 (Figure 4), in which there are neither process crashes nor wrong suspicions. The simplicity of the MR algorithm design and the low number of communication steps (only two) is the reason behind its very good performance in the multicast model (Figure 4a) and even in the unicast model (Figure 4b) which favors centralized communication patterns since, in this model, multicast messages are sent one by one, causing greater contention within the network. The same explanation holds for the FLC algorithm because it has the same structure as the MR, except that FLC also has the election phase which adds a communication step to the algorithm. This additional communication step explains the fact that the MR algorithm performs better in this scenario than the FLC. The MRLeader algorithm exchanges the greatest number of messages, causing more contention in the network, thus explaining its poor performance. The FLC algorithm produces less contention in the network than the MRLeader algorithm because the first phase of the FLC algorithm (leader election phase) costs only $O(n)$ messages while each phase of the MRLeader costs $O(n^2)$. The CT and Paxos algorithms have the same structure, except that Paxos also has a prepare message at the beginning of the algorithm, which justifies the same performance in this scenario.

We now look at the second scenario in a multicast network (Figure 5) where there are crashed processes, but no wrong suspicions. The major factor affecting the performance in this situation is how the algorithm chooses the coordinator/leader process. In a coordinator-based algorithm the coordinator changes at each round and thus, in case of multiple simultaneous crashes of coordinator processes, the algorithm is forced to start a new round even if its *coordinator* has crashed, until it reaches a round whose coordinator is correct. In contrast, in a leader-based algorithm, the leader oracle directly gives the algorithm the index of a valid (correct) process to lead the next round and, as a result, the algorithm avoids starting rounds for already crashed processes. This explains why the leader-based algorithms (MRLeader, Paxos and FLC) perform better than the coordinator-based ones (MR and CT) in the case of 3 crashes (Figure 5c). The FLC algorithm performs better than the Paxos algorithm due to its decentralized communication pattern that allows for any process to decide, once it receives a majority of non null estimates during the second phase of the algorithm, while at the same time the decentralized pattern takes advantage of the multicast network when sending a message to multiple destinations. As explained for the first scenario, the MRLeader algorithm produces more contention



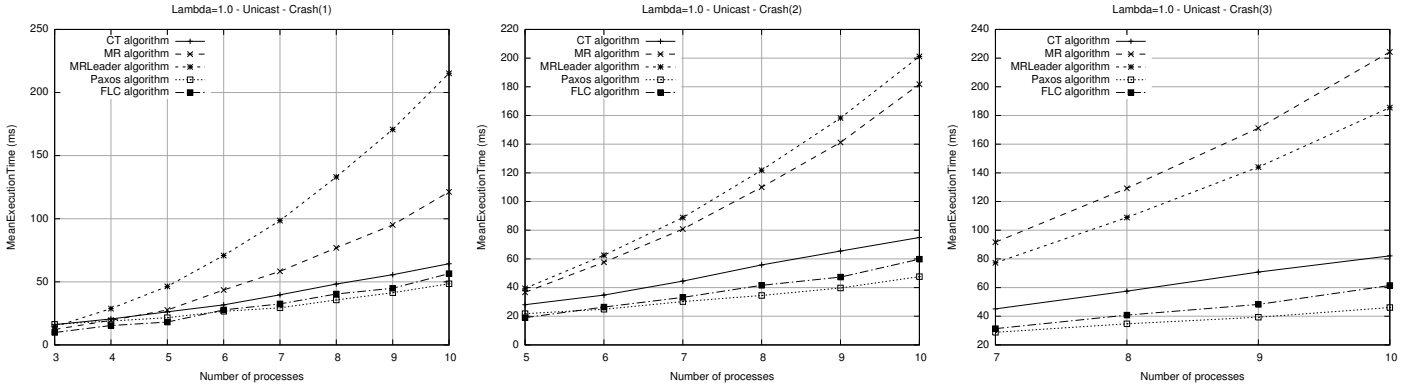
(a) Multicast network (b) Unicast network

Figure 4: Performance Results for Scenario 1 (no crashes, no wrong suspicions)



(a) Crash of p_0 (b) Crash of p_0, p_1 (c) Crash of p_0, p_1, p_2

Figure 5: Performance Results for Scenario 2 (multiple crashes, no wrong suspicions) - Multicast Network



(a) Crash of p_0 (b) Crash of p_0, p_1 (c) Crash of p_0, p_1, p_2

Figure 6: Performance Results for Scenario 2 (multiple crashes, no wrong suspicions) - Unicast Network

in the network than the FLC algorithm, which justifies the fact that FLC performs better than the MRLeader algorithm. The CT algorithm generates more contention in the network because all processes send a *nack* message to the crashed coordinator before passing to the next round, a contention proportional to the number of processes in the system.

The centralized communication pattern algorithms (Paxos and CT) win the battle for the unicast network model in Scenario 2, except that the FLC algorithm surprisingly outperforms the CT algorithms (Figure 6). This is due to the

contention caused by the CT algorithm when sending the *nack* messages to the crashed coordinator before moving on to the next round. In addition, the first phase of the FLC algorithm that costs only $O(n)$ messages is suitable to the unicast network model and helps the FLC algorithm to give a good performance in this network model. The Paxos algorithm performs the best in all three cases because it has a centralized communication pattern and is leader-based.

Generally speaking, we can say that the FLC algorithm gives the best performance when there are process crashes in a multicast network. Even when we use a unicast network model with process crashes the FLC algorithm gives a very acceptable performance by outperforming the MR, MRLeader and CT algorithms. In addition, when processes do not crash (Scenario 1) the FLC algorithm directly follows the MR algorithm, outperforming the CT, Paxos and MRLeader algorithms. When comparing the FLC algorithm only to the MRLeader algorithmsince they share the same classification (both being decentralized and leader-based)it is to be noted that the FLC gives the better performance of the two scenarios and of the two network models (multicast and unicast).

IV. CONCLUSION AND FUTURE WORK

We have presented a new leader-based consensus algorithm (the FLC algorithm) for the crash-stop failure model. In addition, we analyzed the performance of our algorithm and compared it to four of the most well-known consensus algorithms in the crash-stop failure model: CT, Paxos, MR and MRLeader. The principal differences between these algorithms are their communication patterns (centralized or decentralized) and the way they choose coordinator/leader processes (coordinator-based or leader-based). The FLC and the MRLeader algorithms share the same characteristics (both are decentralized and leader-based), yet the performance results show that the FLC algorithm outperforms the MRLeader algorithms in all the proposed scenarios. The results also show that the FLC algorithm is better than the other four algorithms in the case of process crashes in a system with a multicast network model. In addition, the leader election phase of the FLC algorithm can tolerate a minority of detection errors even if they happen in the eventual leader process.

In our domain of application of consensus problems (Managing data consistency in collaborative systems in the context of high process mobility) we are mainly interested in the performance of consensus algorithms in the presence of multiple process crashes with a multicast network model. The FLC algorithm is therefore the best solution in this case especially because its performance is highly acceptable when no process crashes take place.

In the future we plan to study the influence of wrong suspicions on the performance of consensus algorithms. Another direction for our work may be to create and conduct experiments with a new version of the FLC algorithm for the crash-recovery failure model.

REFERENCES

[1] L. Droz-Bartholet, J.-C. Lapayre, F. Bouquet, É. Garcia, and A. Heinisch, "Ramos: Concurrent writing and reconfiguration for collaborative systems," *J. Parallel Distrib. Comput.*, vol. 72, no. 5, pp. 637–649, 2012.

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

[3] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[4] A. Mostéfaoui and M. Raynal, "Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach," in *Proceedings of the 13th International Symposium on Distributed Computing*. London, UK, UK: Springer-Verlag, 1999, pp. 49–63.

[5] ——, "Leader-based consensus," *Parallel Processing Letters*, vol. 11, no. 1, pp. 95–107, 2001.

[6] P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," *Journal of Information Science and Engineering*, vol. 18, no. 6, pp. 981–997, November 2002.

[7] A. Coccoli, P. Urban, A. Bondavalli, and A. Schiper, "Performance analysis of a consensus algorithm combining stochastic activity networks and measurements," in *Proceedings of International Conference on Dependable Systems and Networks, 2002. DSN 2002.*, 2002, pp. 551 – 560.

[8] N. Sergent, X. Defago, and A. Schiper, "Impact of a failure detection mechanism on the performance of consensus," in *Proceedings of Pacific Rim International Symposium on Dependable Computing, 2001.*, 2001, pp. 137 –145.

[9] L. Sampaio, M. Hurfin, F. Brasileiro, and F. Greve, "Evaluating the impact of simultaneous round participation and decentralized decision on the performance of consensus," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07.*, 2007, pp. 625–634.

[10] P. Urbán and A. Schiper, "Comparing distributed consensus algorithms," in *Proceedings of International Conference on Applied Simulation and Modelling*, 2004, pp. 474–480.

[11] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama, "Performance comparison of a rotating coordinator and a leader based consensus algorithm," in *SRDS*. IEEE Computer Society, 2004, pp. 4–17.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[13] R. Guerraoui and A. Schiper, "Consensus: the big misunderstanding [distributed fault tolerant systems]," in *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 1997.*, oct 1997, pp. 183 –188.

[14] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, no. 1, pp. 77–97, Jan. 1987.

[15] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[16] R. Guerraoui, M. Hurfinn, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, "Consensus in asynchronous distributed systems: A concise guided tour," in *Advances in Distributed Systems*, ser. Lecture Notes in Computer Science, S. Krakowiak and S. Shrivastava, Eds. Springer Berlin Heidelberg, 2000, vol. 1752, pp. 33–47.

[17] B. Charron-Bost, "Agreement problems in fault-tolerant distributed systems," in *SOFSEM 2001: Theory and Practice of Informatics*, ser. Lecture Notes in Computer Science, L. Pacholski and P. Ružička, Eds. Springer Berlin Heidelberg, 2001, vol. 2234, pp. 10–32.

[18] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, Jul. 1996.

[19] L. Lamport, "Paxos made simple," *SIGACT News*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[20] P. Urbán, "Evaluating the performance of distributed agreement algorithms: Tools, methodology and case studies," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, Switzerland, August 2003, number 2824.

[21] F. Hanna, L. Droz-Bartholet, and J.-C. Lapayre, "The proof of the flc consensus algorithm," Femto-st/DISC, Tech. Rep. RTDISC2014-1, 2014. [Online]. Available: http://members.femto-st.fr/sites/femto-st.fr.fouad-hanna/files/content/flc/RTDISC2014-1.pdf

[22] (2014, March). [Online]. Available: http://members.femto-st.fr/fouad-hanna/en/flc-algorithm

[23] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Trans. Comput.*, vol. 51, no. 5, pp. 561–580, May 2002.

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