One-step Consensus with Zero-Degradation

Dan Dobre and Neeraj Suri
Department of Computer Science, Technische Universität Darmstadt
{dan,suri}@informatik.tu-darmstadt.de

Abstract

In the asynchronous distributed system model, consensus is obtained in one communication step if all processes propose the same value. Assuming $f < n/3$, this is regardless of the failure detector output. A zero-degrading protocol reaches consensus in two communication steps in every stable run, i.e., when the failure detector makes no mistakes and its output does not change. We show that no leader-based consensus protocol can be simultaneously one-step and zero-degrading. We propose two approaches to circumvent the impossibility result and present corresponding consensus protocols. Further, we present an atomic broadcast protocol that has a latency of 3δ in every stable run and a latency of 2δ in case of no collisions. Finally, we evaluate its performance in a cluster of workstations.

1 Introduction

Consensus is central to the construction of fault-tolerant distributed systems. Atomic broadcast, which is at the core of state machine replication [20] can be implemented as a sequence of consensus instances [5]. As this approach requires solving consensus repetitively, performance of consensus becomes a crucial issue. We measure performance by counting the number of communication steps needed to achieve consensus. Actually, in asynchronous systems we cannot limit the number of communication steps, since this would contradict the well-known FLP impossibility result [10]. In practice however most runs of a distributed system are failure-free and synchronous. If consensus is used in a repeated form, then the overhead caused by runs with failures is negligible. Since failures that occur in one run propagate as initial failures to all subsequent runs, we are interested in algorithms whose performance is not permanently[1] affected by initial failures. We say that a run of a consensus algorithm is *stable* iff the failure detector makes no mistakes and its output does not change during that run. As defined in [9], a *zero-degrading* algorithm achieves consensus in *two* communication steps in every stable run.

[1]Since failure detection is not instantaneous, we cannot require that a run is not affected by initial failures. Such *recovery* runs [22] have negligible impact on the overall performance of a repetitive consensus execution.

Another important optimization aspect is to expedite the decision when all processes propose the same initial value. Assuming $f < n/3$, no underlying failure detector is needed and *one* communication step is sufficient to obtain consensus. In the literature such algorithms are called *one-step*. The original idea of consensus in one communication step stems from Brasileiro [2]. While his solution is optimal for this particular case, his protocol needs at least three communication rounds starting from other configurations.

Our first objective is to investigate if consensus protocols are inherently either one-step or zero-degrading. The question we ask is the following: Do one-step consensus protocols need *three* communication steps in general? In section 4 we show that no leader-based consensus protocol can be simultaneously one-step and zero-degrading. This implies that every one-step protocol based on leader election has a failure-free and synchronous run in which some process decides after three communication rounds or later [11].

Our second objective is to find sufficient conditions for circumventing the established impossibility result and hence to eliminate the overhead of one communication step. In this paper we consider two different approaches and present corresponding consensus protocols. In the first approach, we condition one-step decision on the behaviour of the failure detector. With this approach, one-step decision is guaranteed only in stable runs. The consensus algorithm we present in section 5 is both of practical and of theoretical interest. It is theoretically appealling because it uses the Ω failure detector, which is the weakest to solve consensus [4]. Moreover, it is reasonable to require stability, as without it even termination cannot be guaranteed. Since stability frequently holds in practice, it is reasonable to optimize in this respect. The second approach has been originally proposed by Lamport [15]. In section 6 we present a modified version thereof based on the $\Diamond\mathcal{P}$ failure detector. This protocol provides both one-step decision and zero-degradation.

Further, in section 7 we present a consensus based atomic broadcast algorithm that has a latency of 3δ in every stable run and a latency 2δ in case of no collisions, where δ is the maximum network delay. In section 8 we present a corresponding analytical and experimental evaluation.

2 Related Work

Brasileiro's [2] one-step consensus algorithm has a preliminary voting phase in which processes exchange their proposals. If a process receives enough equal values it decides, otherwise it uses an underlying consensus module. If some process decides v after the first step, all processes that proceed without deciding propose v to the consensus module. Agreement is thus ensured by the properties of the underlying consensus. The drawback of this algorithm is that it needs three rounds from other initial configurations.

Based on Brasileiro's idea, Mostefaoui and Raynal [17] developed an atomic broadcast protocol that has two message delays in the best case but needs four in the normal case. Moreover, even if messages are ordered, it is very unlikely that all buffers have the same length when their content is proposed. Thus, distinct processes propose different values and the protocol works in the slower mode.

This problem was recognized by Pedone and Schiper [18] and they suggested agreement on the largest common prefix instead of agreement on the whole buffer. As long as all buffers share a nonempty common prefix of messages, their algorithm achieves a latency of two message delays. As soon as messages are out of order, consensus is needed, which adds a latency of two additional message delays. This protocol tolerates a minority of faulty processes, but achieving a latency of 2δ requires collecting the proposals from *all* processes. Thus, even if a single process crashes, the protocol switches to the slower mode.

Based on the observation that in LANs, messages are frequently delivered in total order, Pedone and Schiper [19] introduced the notion of *ordering oracle* to model the spontaneous total order encountered in LANs. The authors present an atomic broadcast protocol that has a latency of two message delays in case of no collisions and thus performs very well for a low to medium throughput. However, for high throughputs and hence with the increase of collisions their solution exhibits a considerable performance degradation.

Recently, the authors of [3] have extended the idea of weak ordering oracles to Paxos-like [13] protocols. Paxos-like protocols allow for the recovery of crashed processes [1] and are well suited for the client/server computation model. The R*-Consensus protocol of [3] degrades if multiple clients issue requests concurrently and thus it suffers from the same drawback as the original [19].

The key assumption in Brasileiro's [2] one-step consensus is $f < n/3$. This is generalized by Lamport [14] who distinguishes between the number of correct processes required to reach consensus in one communication step ($n-e$ with $e \leq f$) and the number of correct processes needed for progress ($n - f$ with $f < n/2$). Intuitively, if a process p decides v in one communication step, then it has received $n - e$ equal values v. Consequently, every process q that receives a message from $n - f$ processes receives v $n - e - f$ times. Since among the $n - f$ values received by q at most e values are distinct from v, agreement is ensured if $n - e - f > e$. Thus, the degree of resilience is given by $n > max\{2f, 2e + f\}$. Maximizing e leads to $f < \lfloor n/3 \rfloor$, while maximizing f leads to $e \leq \lfloor n/4 \rfloor$.

Recently, Lamport has presented Fast Paxos [15], an extension to the classic Paxos [13] consensus protocol, that meets all lower bounds on time-complexity and resilience. Fast Paxos switches between achieving consensus in two or three message delays depending on how many processes are working. If $n - f$ processes are working, then Fast Paxos achieves consensus in three message delays. If $n - e$ processes are working and there are no concurrent proposals, then Fast Paxos obtains consensus in two message delays.

3 System Model and Definitions

We assume a crash-stop asynchronous distributed system model [5] consisting of a set $\Pi = \{p_1, ..., p_n\}$ of n processes of which up to $f < n$ may fail by crashing. A process behaves correctly, i.e., it executes the algorithm assigned to it until it possibly crashes. A process that never crashes is *correct*, otherwise it is *faulty*. Message transmission delays and relative processing speeds are unbounded. The absence of timing assumptions makes the distributed system *asynchronous* [16]. Processes communicate and synchronize by sending and receiving messages over *reliable* channels. A reliable channel does not lose, duplicate or (undetectably) corrupt messages. Given two correct processes p and q, if p sends a message m to q then q eventually receives m.

3.1 The Consensus Problem

In the consensus problem, a set of processes have to agree unanimously on a value that is one of the values proposed by some process from the set. Formally, consensus is defined by two safety properties (Validity and Agreement) and one liveness property (Termination) [5]:

Validity: If a process decides v, then some process has proposed v.

Agreement: No two processes decide differently.

Termination: Every correct process decides.

Asynchrony and crashes create a context in which consensus has no deterministic solution [10]. Various approaches have been proposed to circumvent this impossibility result. In practice, distributed systems are synchronous most of the time so that models such as partial synchrony [8], the timed asynchronous model [7] and unreliable failure detectors [5] describe real systems more accurately than the asynchronous model.

3.2 Failure Detectors

Instead of dealing with low level details about synchrony and associated timing assumptions, failure detectors [5] are defined in terms of properties, allowing a clean separation from the implementation. We assume that the system is equipped with an appropriate distributed failure detector, consisting of one failure detector module installed at each process. The consensus protocols presented in this paper use the Ω and $\diamond\mathcal{P}$ failure detectors respectively. Both *eventually* provide consistent and correct information about the state of processes, i.e., crashed or not crashed. While $\diamond\mathcal{P}$ tracks the state of every process, Ω eventually outputs a single correct *leader* process. Ω is strictly weaker than $\diamond\mathcal{P}$ and it is the weakest failure detector to solve consensus [4, 6]. $\diamond\mathcal{P}$ is defined in terms of the following two properties:

$\diamond$ **Strong Completeness:** Eventually, every crashed process is suspected by every correct process.

$\diamond$ **Strong Accuracy:** Eventually, no correct process is suspected by any correct process.

Ω is defined in terms of the eventual leadership property:

Eventual Leader: Eventually, Ω outputs the same correct process forever.

3.3 The Atomic Broadcast Problem

In the atomic broadcast problem processes have to agree on an unique sequence of messages. Formally, the atomic broadcast problem is defined in terms of two primitives a-broadcast(m) and a-deliver(m), where m is some messsage. When a process p executes a-broadcast(m) (respectively a-deliver(m)), we say that p a-broadcasts m (respectively p a-delivers m). We assume that every message m is uniquely identified and carries the identity of its sender. In this context, the atomic broadcast problem is defined by two liveness properties (Validity and Agreement) and two safety properties (Integrity and Total Order) [5]:

Validity: If a correct process a-broadcasts a message m, then it eventually a-delivers m.

Agreement: If a process a-delivers message m, then all correct processes eventually a-deliver m.

Integrity: For any message m, every process a-delivers m at most once, and only if m was previously a-broadcast.

Total Order: If some process a-delivers message m' after message m, then a process a-delivers m' only after it a-delivers m.

3.4 Spontaneous Order

As pointed out by Pedone and Schiper in [19], messages broadcast in LANs are likely to be delivered totally ordered. This phenomenon can be attributed to the short delay between the broadcast of a message and the subsequent delivery. Consequently, if two distinct processes broadcast m and m' respectively, then it is very likely that m is delivered by all processes before m' or viceversa. The authors of [19] propose a new oracle called *Weak Atomic Broadcast* (WAB) that captures the property of spontaneus total order. A WAB is defined by the primitives w-broadcast(k,m) and w-deliver(k,m), where $k \in \mathbb{N}$ is the k-th w-broadcast instance and m is a message. When a process p executes w-broadcast(k, m), we say that p w-broadcasts m in instance k. When a process p executes w-deliver(k, m) we say that p w-delivers m that was w-broadcast in instance k. Intuitively, if WAB is invoked infinitely often, it gives the same output to every process infinitely often. Formally, a WAB oracle satisfies the following properties:

Validity: If a correct process invokes w-broadcast(k, m), then all correct processes eventually get the output w-deliver(k, m).

Uniform Integrity: For every pair (k,m), w-deliver(k,m) is output at most once and only if some process invoked w-broadcast(k,m)

Spontaneous Order: If w-broadcast(j,$*$) is called an infinite number of instances j then there are infinite k such that the first message w-delivered in instance k is the same for every process that w-delivers messages in k.

4 Lower Bound Proof

In this section we prove a lower bound on consensus time-complexity. We show that every one-step leader-based protocol has a run in which some process needs at least three communication steps to decide. In other words it is impossible to devise a leader-based consensus protocol that is one-step and zero-degrading. In order to develop an intuition for the impossibility result, we first describe Brasileiro's one-step consensus [2] and how we would have to combine it with a leader-based protocol to achieve zero-degradation.

In the first round of Brasileiro's one-step consensus, every process broadcasts its proposal and subsequently waits for a message from $n - f$ processes. A process p decides v iff it receives $n - f$ equal values v. Hence if a process p decides v, then every process q necessarily receives v at least $n - 2f$ times. To ensure agreement, it is sufficient to require that v is a *majority* among the values received by q.

If there are less than $n - f$ equal proposals, then the first round is wasted. To eliminate this overhead, one straightforward approach is to combine it with the first round of a leader-based protocol. Here, consensus is obtained in two communication steps if every correct process picks the leader value in the first round. Hence, in the combined protocol we have to ensure that if no process decides in the first round, then every correct process picks the leader value. However, this is only possible if there are less than $n - 2f$ equal proposals. Otherwise, it might happen that some process receives a majority value v and consequently picks v in order to ensure agreement while some other process picks the leader value v_l and $v \neq v_l$. Hence, two distinct values are proposed in the second round and consequently some process might not decide before the third round.

Definition 1 (one-step) *Assuming $f < n/3$, a consensus protocol is* one-step *iff it reaches consensus in one communication step in every run in which all proposals are equal.*

Definition 2 (stable run) *A run of a consensus algorithm is* stable *iff the failure detector makes no mistakes and its output does not change during that run.*

The stability of the failure detector can be attributed to the fact that nearly all runs are synchronous and crashes are initial. Even if the failure detector needs to pass through a temporary stabilization period (e.g. after a failure), in most runs it will exhibit a stable and accurate behaviour. In a stable run, Ω outputs the same correct process from the beginning of the run, while $\Diamond\mathcal{P}$ suspects exactly the processes that have crashed initially.

Definition 3 (zero-degradation) *A consensus algorithm $\mathcal{A}$ is* zero-degrading *iff $\mathcal{A}$ reaches consensus in two communication steps in every stable run.*

Theorem 1 (Lower Bound) *Assuming that $n/4 \leq f < n/3$, every one-step consensus algorithm $\mathcal{A}$ based on Ω has a stable run in which some process decides after three communication steps or more.*

Preliminary notes (see Figure 1): We prove the theorem for the case $n = 4$ but this solution can be generalized to any value of n by employing the same technique as used in [11]. The state of a process after k communication steps is determined by its initial value, the failure detector output and the value and source of the messages received in every communication round up to k. To strengthen the result, the processes exchange their complete state. For the sake of simplicity, Ω outputs the same leader process p_1 at all processes in every run considered in the proof until p_1 possibly crashes. The state of process p after k communication steps is expressed as a k-dimensional vector with n entries such that the i-th entry contains the state of the i-th process after $k-1$ steps. Since in each round a process waits for a message from at most $n-f$ processes, one entry is empty. The decision value is bracketed (0)/(1).

Two runs R_1 and R_2 are *similar* for process p up to step k, iff the state of p after k steps in R_1 is identical to the state of p after k steps in R_2. If two runs are similar for some process p, then p decides the same value in both runs. *Idea*: The proof is by contradiction. We assume a leader-based one-step and zero-degrading protocol and show that it does not solve consensus. We construct a chain of legal runs such that every two neighboring runs are similar to some process. We start with a run in which all processes propose 1, and then we construct new runs either by changing the communication pattern or the configuration. The failure detector assumption as well the expected properties of the protocol lead to a run that violates agreement.

Proof:

- If $\mathcal{A}$ is one-step, then it must have a run like R_1 in which all correct processes propose 1 and p_1 might have proposed the same. Thus, p_4 decides 1 after one round[2].
- If $\mathcal{A}$ is zero-degrading, then it must allow a run such as R_2. R_2 is stable because Ω outputs p_1 at all correct processes and its output does not change. Thus, p_1 decides

[2]Actually, processes p_2 and p_3 also decide 1 after one round but this is not relevant for the proof.

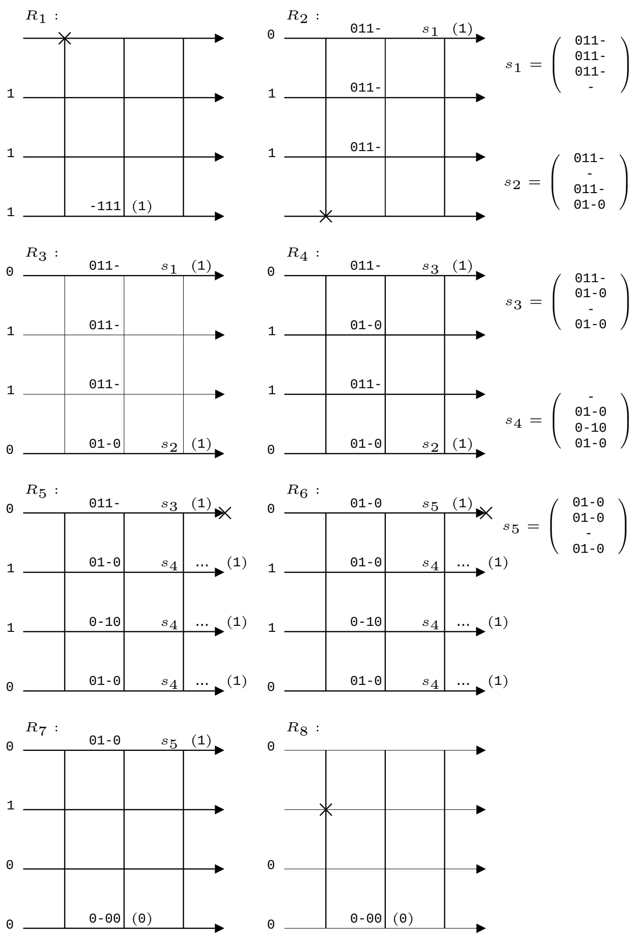

Figure 1: Lower bound proof.

after two communication steps. If p_1 decides 0, then we can construct a run R' that for p_1 is similar to R_2 (p_1 decides 0 in R') and that for p_4 is similar to R_1 (p_4 decides 1 in R'). Consequently in R_2, p_1 necessarily decides 1.
- Runs R_2 and R_3 are similar for p_1. Thus, p_1 decides 1 in R_3 after two communication steps. Since R_3 is stable, p_4 also decides 1 after the second round.
- Runs R_3 and R_4 are similar for p_4 and thus p_4 decides 1 in R_4 after the second communication step. Since R_4 is stable, p_1 also decides 1 after the second round.
- Runs R_4 and R_5 are similar for p_1. Consequently p_1 decides 1 in R_5 after the second communication step. In R_5 we crash p_1 so that all messages sent to p_2, p_3 and p_4 after the first round are lost. Since R_5 is not stable because Ω eventually outputs a new leader, p_2, p_3 and p_4 are only required to decide eventually. In order to ensure agreement they eventually decide 1.
- In R_6 we crash p_1 such that R_5 and R_6 are similar for p_2, p_3 and p_4. Thus, they eventually decide 1. As p_1 cannot distinguish R_6 from a stable run, it decides after two rounds. In order to ensure agreement, p_1 necessarily

decides 1.

- R_6 and R_7 are similar for p_1. Thus, p_1 decides 1 in R_7 after two communication steps. R_7 and R_8 are similar for p_4, thus it decides 0 in $R7$ after the first communication step. Consequently, R_7 violates the consensus agreement property concluding the proof. □

5 Circumventing the Impossibility with Ω

In this section we present a leader-based consensus protocol that is zero-degrading but is not one-step, as this would contradict the established impossibility result. However, the protocol has the property that it obtains consensus in one communication step if all proposals are equal *and* the run is stable. The main idea behind the proposed **L**-Consensus algorithm 1 is to constrain the processes to decide the value proposed by the leader. A process decides v in the first round if $n-f$ values including the leader value are equal to v. Consequently, every process that does not decide can safely pick the leader value. Hence, consensus is achieved in two rounds in every stable run. If there is no leader, then safety is ensured by picking the majority value.

Algorithm 1: The **L**-Consensus Algorithm

```
Function Consensus($v_i$)
   start $T1$, $T2$;
   task $T1$: $r_i \leftarrow 1$; $est_i \leftarrow v_i$; $ld \leftarrow \perp$;
      while true do
         $ld \leftarrow \Omega.leader$;
1        $\forall j$ do send PROP($r_i$, $est_i$, $ld$) to $p_j$ enddo;
2        wait until received PROP($r_i$, $*$, $*$) from $n-f$ processes;
3        wait until received PROP($r_i$, $*$, $*$) from $ld$
         $\vee$ $ld \neq \Omega.leader$;
4        if received PROP($r_i$, $v$, $ld$) from $n-f$ processes $\wedge$ received
         PROP($r_i$, $v$, $*$) from $ld$ then
5           $\forall j$ do send DECIDE($v$) to $p_j$ enddo;
6           return $v$;
7        else if received PROP($r_i$, $*$, $ld$) from $> n/2$ processes $\wedge$
         received PROP($r_i$, $v$, $*$) from $ld$ then
8           $est_i \leftarrow v$;
9        else if received PROP($r_i$, $v$, $*$) from $n-2f$ processes then
10          $est_i \leftarrow v$;
         $r_i \leftarrow r_i + 1$;
      end
11 task $T2$: upon reception of DECIDE($v$): $\forall j \neq i$ do send
   DECIDE($v$) to $p_j$ enddo; return $v$;
```

The protocol executes in a round by round fashion. In every round, processes exchange messages, update their state depending on the messages received and possibly decide or move to the next round. The algorithm has three blocks that a process can execute in a round depending on which condition is satisfied (at line 4, 7 or 9). Safety is ensured as follows: if a process p decides a value v during round k, every process q that finishes round k, does so with value v, no matter what block it executes. In a stable run, the condition at line 7 evaluates to true, every correct process accepts the leader value and hence decides in the next round. In asynchronous runs, when there might be multiple leaders in the system, agreement is kept through majority voting. Since $n-f$ equal values are necessary for a decision, if a process decides v then every process receives v at least $n-2f$ times, making the condition at line 9 become true. Since $n-2f > f$, a process can safely pick the majority value.

5.1 Detailed Description

The **L**-Consensus algorithm consists of two parallel tasks $T1$ and $T2$. When a process p_i calls the `Consensus` function with a proposal v_i (i.e. it proposes value v_i), it initiates both tasks. Compliant with the definition of consensus, the `Consensus` function eventually returns the same decision value v to each non-crashed process.

Task 1: The algorithm executes a sequence of asynchronous rounds of one communication step each. In each round k, a process sends a round k message containing its current proposal to all processes and waits for round k messages from $n-f$ processes including its current leader, computes its new state based on the messages received (possibly decides), and moves to the next round. A process p_i maintains three local variables: the round number r_i, an estimate of the decision value est_i initialized to the proposal value v_i, and the current leader ld, initially $\perp$.

At the beginning of each round, p_i queries Ω for the current leader and stores the identity in ld. We say that p_i has leader p_l in round k if p_i sends a message with $ld = l$. The messages sent contain the following fields: k_i, est_i, ld. We say that a process p_l is *majority leader* for round k if a majority of processes send round k messages with $ld = l$. As any two majorities have a non empty intersection, there can be at most one majority leader at round k. Note that in asynchronous runs there are periods with no majority leader.

A process p_i can send two different types of messages in round k. If p_i has decided, then it broadcasts a decision value, otherwise it broadcasts a PROP(k_i, est_i, ld) message and we say that p_i proposes est_i in round k_i.

At the end of round k (i.e. after receiving round k messages from $n-f$ processes possibly including one from ld), process p_i updates its est_i variable as follows: if p_i receives a value v from the majority leader of round k, then $est_i = v$. If there is no majority leader or the Ω module at p_i suspects ld for having crashed and p_i receives $n-2f$ equal values v, then p_i picks v. Otherwise the estimate value is kept unchanged. A process p_i decides in round k if it receives $n-f$ equal values including one value from the majority leader.

Task 2: Upon receiving a decision message with value v, p_i forwards the decision value to the other processes and then decides v. Thus, if a correct process decides, the remaining correct processes cannot block since they eventually receive the decision message.

5.2 Correctness

Lemma 1 (Termination) *Every correct process decides.*

Proof: We show that if some correct process never decides

then every correct process eventually decides; a contradiction. If some correct process never decides then either some correct process decides or no correct process decides.
1) Case a: Some correct process decides. Then, it broadcast a decision message (line 5). Since it is correct, every correct process eventually receives the decision message (line 16) and also decides. Thus, every correct process decides, which contradicts the assumption.
2) Case b: No correct process decides. If some correct process p_i never decides, then either it is blocked in a round or it executes an infinite number of rounds.

Case 1: p_i blocks forever in a round. Let k be the first round in which some correct process is blocked. p_i can only be blocked at one of the wait statements (line 2 or 3).
- Case I: p_i is blocked at line 2 of round k. Since k is the first round in which some correct process blocks at line 2, all correct processes have broadcast a round k message at line 1. As communication links are reliable and there are at least $n - f$ correct processes, p_i eventually receives $n - f$ round k messages and completes line 2.
- Case II: p_i is blocked at line 3 of round k. As in the case above, every correct process broadcasts a round k message. Consider ld, which is the leader process output by Ω at p_i. If ld is correct, then p_i eventually receives a round k message from ld and completes line 3. Otherwise, if ld is faulty, then either p_i eventually receives a round k message from ld, or Ω eventually outputs a correct process different from ld and p_i completes line 3. Thus, p_i cannot block at line 3.

Case 2: All correct processes execute an infinite number of rounds without deciding. From the definition of a faulty process, there is a time t_1 such that every faulty process has crashed before t_1. From the definition of Ω there is a time t_2 such that Ω outputs the same correct process p_l at every correct process forever. Let $t := max\{t_1, t_2\}$ and k be the first round after t. In round k, every correct process sets ld to l and sends a message (k, $*$, l) to all processes. Since no correct process decides, no correct process executes line 5. As there is a majority of correct processes and p_l is not suspected by any correct process, every correct process receives a majority of round k messages including one message from p_l, and every correct process sets its est variable to the same value (line 8). Therefore, at round $k + 1$ every process including p_l sends a ($k + 1$, v, l) message. Thus, at round $k + 1$ every correct process receives $n - f$ equal messages including a ($k + 1$, v, l) message from p_l. Therefore, the condition at line 4 evaluates to true and every correct process decides at line 5; a contradiction. □

Lemma 2 (Agreement) *No two processes decide differently.*

Proof: A process can decide either at line 5 of some round or at line 16 of task $T2$. If a process decides v at line 16, then some other process has decided v at line 5. Let k be the lowest round in which some process p decides v at line 5. We claim that each process that decides v at line 5 of round k decides v, and that every process that completes round k does so with $est = v$. This implies that the est value of every process after round k is always v. Thus, in round k and after round k, v is the only value that can be decided at line 5. As k is the lowest round in which some process decides, this implies that v is the only value that can be decided in a round at line 5. This also implies that no process decides a value different from v at line 16 of task $T2$. Now we prove the above claim. Suppose that a process $q \neq p$ decides d in round k. Since $n - f > n/2$, both p and q receive equal values v and d respectively from a majority of processes. As any two majorities intersect in at least one process, it follows that $d = v$. Now, consider any process q' that completes round k without deciding. We show that q' completes round k with $est = v$. There are two cases to consider: Case 1: q' evaluates the condition at line 7 to false. We show that q necessarily evaluates the condition at line 9 to true. At round k there are at least $n - f$ values v and q' has received $n - f$ values at line 2 of round k. Any two sets of $n - f$ elements have $n - 2f$ elements in common, thus among the $n - f$ values q' receives at round k, at least $n - 2f$ values are equal to v and at most f values are distinct from v. Since $n - 2f > f$, v is a majority value among the values received by q'. Value v is unique as there cannot be two distinct majority values. Thus q' completes round k with $est = v$.

Case 2: q' evaluates the condition at line 7 to true. Thus, there must be a process p_l such that a majority of processes send messages with $ld = l$. Since p decides in round k, there must be a process $p_{l'}$, such that $n - f$ processes send messages with $ld = l'$. As any two majorities have a process in common, it follows that $l = l'$. Thus q completes round k with $est = v$. □

6 Circumventing the Impossibility with $\diamond\mathcal{P}$

In this section we present a one-step and zero-degrading algorithm that uses the $\diamond\mathcal{P}$ failure detector. The proposed **P**-Consensus algorithm 2 is based on a simple observation that was originally discovered by Lamport [15]. One of the necessary conditions for the impossibility of section 4 is that processes receive messages from different quorums in the first communication round. If all processes received the same set of messages, then they could deterministically pick the same value to propose in the second round. Consequently, consensus is obtained in two communication steps.

The idea behind **P**-Consensus is to use the $\diamond\mathcal{P}$ failure detector to build a consistent quorum from which every process delivers first round messages in case it cannot decide. In every stable run, $\diamond\mathcal{P}$ suspects exactly the faulty processes and its output does not change during that run. Hence, every process that does not decide during the first round computes the same quorum (line 5) and subsequently receives a message from every quorum member. The sets of messages received by different processes from the quorum are equal and the functions applied to pick a value are deterministic (lines 9-12). Hence, all processes start the next round with the same value and consequently every correct process decides in the second round.

Algorithm 2: The **P**-Consensus Algorithm

```
   Function Consensus($v_i$)
     start $T1, T2$;
     task $T1$: $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
       while true do
1          $\forall j$ do send PROP($r_i$, $est_i$) to $p_j$ enddo ;
2          wait until received PROP($r_i$, $*$) from $n-f$ processes;
3          if received PROP($r_i$, $v$) from $n-f$ processes then
4              $\forall j$ do send DECIDE($v$) to $p_j$ enddo; return $v$;
5          let $Q_i = \{$ the first $n-f$ processes
           $p_j : j \notin \diamond\mathcal{P}.suspected\}$;
6          wait until received PROP($r_i$, $*$) from every
           $p_j : j \in Q_i \backslash \diamond\mathcal{P}.suspected$;
7          let $Qlist_i = (v \mid$ PROP($r_i$, $v$) has been received from
           $p_j : j \in Q_i)$;
8          if $|Qlist_i| = n-f$ then
9              if $\exists v \in Qlist_i : \#(v) \geq n-2f$ then
10                 $est_i \leftarrow v$;
11             else
12                 $est_i \leftarrow est_{min\{j|j \in Q_i\}}$;
           else %ensure agreement%
13             let $vlist_i = (v \mid$ PROP($r_i$, $v$) has been received);
14             if $\exists v \in vlist_i : \#(v) > |vlist_i|/2$ then
15                 $est_i \leftarrow v$;
           $r_i \leftarrow r_i + 1$;
       end
16   task $T2$: upon reception of DECIDE($v$): $\forall j \neq i$ do send
     DECIDE($v$) to $p_j$ enddo; return $v$;
```

6.1 Detailed Description

The **P**-Consensus algorithm consists of two parallel tasks $T1$ and $T2$ that are initiated when a process proposes a value. The `Consensus` function eventually returns the same decision value to every correct process. Since the second task is identical to task $T2$ of the **L**-Consensus protocol, we confine ourselves to describing task $T1$.

The algorithm executes a sequence of asynchronous rounds of one communication step each. In each round k, a process sends a round k message containing its current proposal to all processes and waits for round k messages from $n-f$ distinct processes, computes its new state based on the messages received and tries to decide. If it cannot decide then it possibly waits for more messages, computes its new state and moves to the next round.

A process p_i maintains two local variables: the round number k_i initialized to 1 and an estimate of the decision value est_i initialized to the proposal value v_i. At the beginning of each round, p_i broadcasts a message that contains the following fields: k_i, est_i. A process p_i can send two different types of messages in round k_i. If p_i has decided, then it broadcasts a decision value, otherwise it sends a PROP(k_i, est_i) message to all processes and we say that p_i proposes est_i in round k_i.

Subsequently, p_i waits for a message from $n-f$ distinct processes. If p_i receives $n-f$ identical values it decides. Otherwise, p_i additionally waits for messages from a quorum Q that is computed deterministically as the set that contains the first $n-f$ nonsuspected processes. We say that Q is *complete* iff it has $n-f$ members.

At the end of round k, p_i updates its est_i variable as follows: if there is a complete quorum Q such that p_i receives a message from each process in Q and there is a majority value v among the $n-f$ values received, then $est_i = v$. If there is no such value v, then no process decided in round k. Thus, p can propose any value in the next round. Subsequently, p picks the estimate of the *leader*, the process with the smallest index among all nonsuspected processes. In case that there is no such process, p simply keeps its estimate. If Q is not complete and there is a majority value v among the values received in round k then $est_i = v$. If no such value exists, then p_i moves to the next round.

6.2 Correctness

Lemma 3 (Termination) *Every correct process decides.*

Proof: We follow the same strategy as in section 5.2 and show that if some correct process never decides then every correct process eventually decides. Assuming that some correct process never decides yields two cases. Either some correct process decides or no correct process decides. The latter case implies that some correct process never decides. Thus, either $1)$ it is blocked in a round or $2)$ it executes an infinite number of rounds.

- Case $1)$: The proof is similar to the one of section 5.2. A process cannot block at one of the wait statements (at lines 2, 6) because at most f processes are faulty and $\diamond\mathcal{P}.suspected$ eventually contains all crashed processes.

- Case $2)$: All correct processes execute an infinite number of rounds without deciding. From the definition of a faulty process, there is a time t_1 such that every faulty process has crashed before t_1. From the definition of $\diamond\mathcal{P}$ there is a time t_2 such that after t_2, $\diamond\mathcal{P}$ outputs exactly the crashed processes forever. Let $t := max\{t_1, t_2\}$ and k be the first round after t. Since no correct process decides, no correct process executes line 4 and every correct process executes lines 5, 6 and 7. As $\diamond\mathcal{P}$ behaves perfectly in round k, every quorum Q contains exactly the correct processes. The fact that Q is complete and identical and every correct process receives a message from every member of Q implies that $Qlist$ is the same at every correct process and that $|Qlist| = n-f$. Hence, the condition at line 8 evaluates to true and all correct processes pick the same value either at line 10 or at line 12. Therefore, in round $k+1$, all correct processes send a message with the same value and hence every correct process receives $n-f$ identical values and consequently decides at line 4; a contradition. $\square$

Lemma 4 (Agreement) *No two processes decide differently.*

Proof: We claim that each process that decides at line 4 of round k decides v, and that every process that completes round k without deciding does so with $est = v$. As already shown in 5.2, if this claim is true then agreement holds. Now, we prove the above claim. It is easy to see that if two distinct processes p and q decide in round k, then they decide the same value v. Let q' be a correct process that does not decide in round k. As q' receives at least $x \geq n - f$ messages, it receives at most f values $w \neq v$. Since $x - f \geq n - 2f > f$, v is a majority among the values received by q' in round k which implies that one of the conditions at line 9 or 14 evaluates to true. Thus, q' completes round k with $est = v$, which concludes the proof. □

7 The Atomic Broadcast Protocol

The proposed **C**-Abcast protocol 3 represents a modification of the WABcast atomic broadcast algorithm of [19]. Like the Chandra & Toueg's (CT) Atomic Broadcast protocol [5], **C**-Abcast reduces atomic broadcast to consensus. It executes a series of consensus instances to determine a single message delivery sequence at all processes. Unlike the CT Atomic Broadcast, **C**-Abcast assumes an underlying consensus module that is very efficient in case that all proposals are equal. In order to exploit the efficiency of the underlying consensus, **C**-Abcast uses a WAB oracle to provide the consensus module with equal input values. When the oracle outputs the same proposal to every process, **C**-Abcast has a latency of two message delays, i.e., 2δ; one for asking the oracle plus one communication step for consensus. In case of collisions, consensus is obtained in two communication steps. Hence, **C**-Abcast has a latency of three message delays, i.e, 3δ in the common case.

The protocol consists of three concurrent tasks. A process can either a-broadcast a message (line 2), a-deliver a message (line 4), or w-deliver a message (line 16). A process p a-broadcasts a message m by including m in a set $estimate_p$. This set contains the messages that have not been yet a-delivered by p. The a-deliver($*$) task executes in a round by round fashion. In round k, process p w-broadcasts the set $estimate_p$ and waits to w-deliver the first value v output by its oracle. Then, p proposes v to the k-th consensus instance and waits for the decision. After it decides, p atomically delivers all messages contained in the k-th decision in some deterministic order, removes from $estimate_p$ every message a-delivered so far and moves to the next round. In order to ensure validity, every message a-broadcast by some correct process must eventually be contained in the proposal of every correct process. Thus, in the third task (line 16), every process p includes in $estimate_p$ all messages w-broadcast so far.

Algorithm 3: The **C**-Abcast Algorithm

```
   Initialization:
1     $k_i \leftarrow 1$; $estimate_i \leftarrow \bot$; $adelivered_i \leftarrow \bot$;
2  a-broadcast(m):
3     $estimate_i \leftarrow estimate_i \cup \{m\}$;
4  a-deliver($*$):
5     while true do
6        w-broadcast($k_i$, $estimate_i$);
7        wait until w-deliver of the first message ($k_i, v$);
8        $msgSet_i \leftarrow$ Consensus($k_i, v$);
9        $adeliver_i \leftarrow msgSet_i - adelivered_i$;
10       deliver all messages in $adeliver_i$ atomically in some
         deterministic order;
11       $adelivered_i \leftarrow adelivered_i \cup adeliver_i$;
12       $estimate_i \leftarrow estimate_i - adelivered_i$;
13       $k_i \leftarrow k_i + 1$;
14       if $estimate_i = \emptyset$ then
15          wait until w-deliver of the first message ($k_i, v$)
            $\vee\ estimate_i \neq \emptyset$
     end
16 upon w-deliver($*$, $v$) of the second, third etc. message of any round
17    $estimate_i \leftarrow estimate_i \cup v$;
```

7.1 Correctness

Lemma 5 states that $\forall k \in \mathbb{N}$, a) if a process delivers the k-th message batch, then every correct process also delivers it and b) that the k-th message batch is the same at every process. From a) and b) we can easily deduce Agreement and Total Order. Validity requires a more detailed proof.

Lemma 5 *For all $k > 0$, every process p and every correct process q, if p executes round k until the end then q executes round k until the end and $adeliver_p^k = adeliver_q^k$.*

Proof: We will prove the lemma by induction over k. First, it is easy to see that every correct process executes round 1 until the end. Due to consensus agreement, if p a-delivers messages in round 1 then $adeliver_p^1 = adeliver_q^1$. Now assume that the lemma holds for all k, $1 \leq k < r$. We first show that if p a-delivers messages in round r then q executes round r until the end. If p a-delivers messages in round r, then p returns from the invocation of Consensus($r, *$) at line 8. Since there is at most a minority of faulty processes, at least one correct process u executes Consensus($r, *$). This implies that u w-broadcasts its estimate at line 6. By the induction hypothesis, if p a-delivers messages in round $r - 1$, q executes round $r - 1$ until the end. Thus, q eventually w-delivers the first message of stage r either a) at line 7 or b) at line 15. Without loss of generality, let $estimate_u$ be the first message w-delivered by q in round r. In both cases q breaks from the corresponding wait statement and executes Consensus(r, $estimate_u$)[3]. By consensus termination, q eventually executes round r until the end.

[3]In case q breaks from the second wait statement (line 15) it does not block at the first wait statement (line 7) because it has already w-delivered the first round r message.

We now show that if p a-delivers messages in round r then $adeliver_p^r = adeliver_q^r$. As shown in the first part of the lemma, q executes round r until the end. Thus, q a-delivers messages in $adeliver_q^r$. Due to consensus agreement $msgSet_p^r = msgSet_q^r$. By the induction hypothesis, $\forall k, 1 \leq k < r : adeliver_p^k = adeliver_q^k \Rightarrow \cup_{k=1}^{r-1} adeliver_p^k = \cup_{k=1}^{r-1} adeliver_q^k$. As $adeliver^r = msgSet^r - \cup_{k=1}^{r-1} adeliver^k$, we get $adeliver_p^r = msgSet_p^r - \cup_{k=1}^{r-1} adeliver_p^k = msgSet_q^r - \cup_{k=1}^{r-1} adeliver_q^k = adeliver_q^r$. □

Lemma 6 (Agreement) *If a process a-delivers message m, then all correct processes eventually a-deliver m.*

Proof: Follows directly from Lemma 5. □

Lemma 7 (Total Order) *If some process a-delivers message m' after message m, then a process a-delivers m' only after it a-delivers m.*

Proof: Follows from lemma 5, the total odering of natural numbers, and the fact that messages within a batch are delivered atomically in a deterministic order. □

Lemma 8 (Validity) *If a correct process a-broadcasts message m, then eventually it a-delivers m.*

Proof: The proof is by contradiction. Suppose that a correct process a-broadcasts m but never a-delivers m. By Lemma 6 no correct process a-delivers m. Consider a process p that a-broadcasts a message m. Consequently, p includes m in $estimate_p$ and thus w-broadcasts m. By the validity property of the ordering oracle, every correct process eventually w-delivers m at line 16 and thus includes m in its $estimate$. Since no correct process adelivers m, no correct process removes m from its $estimate$ at line 12. There is a time t so that all faulty processes have crashed before t and at which m is included in the $estimate$ of every correct process. Let k be the lowest round number after t. Every correct process w-broadcasts m in k, which implies that every value proposed to the k-th consensus instance necessarily contains m. Due to validity of consensus, m is included in the $msgSet$ of every correct process. Thus, m is a-delivered by every correct process at round k; a contradiction. □

8 Performance Evaluation

In this section we provide a brief comparison both analytical and experimental to outline the efficiency of our protocols compared to Paxos and WABcast. Table 1 compares the proposed protocols with Paxos [13] and WABCast [19] in terms of time complexity (where δ is the maximum network delay), message complexity, resilience, and the oracle used for termination. In case of no collisions, WABCast as well as **L**-/**P**-Consensus have the same time and message complexity. Compared to Paxos, they trade the maximum degree of resilience, i.e., $f < n/2$ for the lower time complexity of 2δ. In periods with collisions, WABCast might not terminate whereas **L**-/**P**-Consensus have the same time complexity as Paxos though with more messages. We expect the proposed protocols to be as efficient in terms of latency as WABCast when collisions are rare and to exhibit a behaviour similar to Paxos when collisions are frequent.

Table 1: Comparison of various atomic broadcast protocols

Protocol	No Collisions ; Collisions		Resil.	Oracle
	latency	#messages		
Paxos	3δ	n^2+n+1	$f < n/2$	Ω
WABCast	2δ ; ∞	n^2+n ; ∞	$f < n/3$	WAB
L-/**P**-Cons.	2δ ; 3δ	n^2+n ; $2n^2+n$		$\Omega/\diamond\mathcal{P}$

8.1 Experimental Evaluation

We compared the performance of the proposed protocols with Paxos and WABCast. We measured the latency of atomic broadcast as a function of the throughput, whereby latency is defined as the shortest delay between a-broadcasting a message m and a-delivering m. We implemented **L**-/**P**-Consensus and **C**-Abcast using the Neko [21] framework. The experiments were conducted on a cluster of 4 identical workstations (2.8GHz, 512MB) interconnected by a 100Mb ethernet LAN. Different consensus algorithms were tested by exchanging the consensus module of **C**-Abcast. The WAB oracle implementation uses UDP packets whereas the rest of the communication is TCP-based. We considered only stable runs in our experiments. In order to capture the performance of the tested protocols during periods with and without collisions, we varied the throughput between $20 msg/s$ and $500 msg/s$. Figure 2 shows

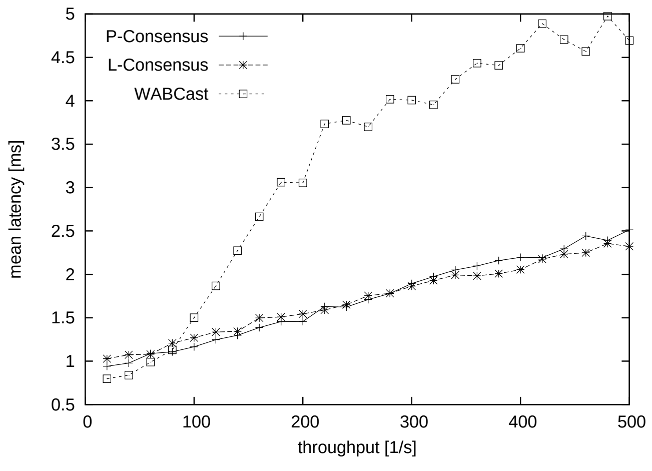

Figure 2: **L**-/**P**-Cons. vs. WABcast ($n = 4$)

the comparison of our protocols with WABCast. Both proposed protocols exhibit a similar latency as WABCast up to

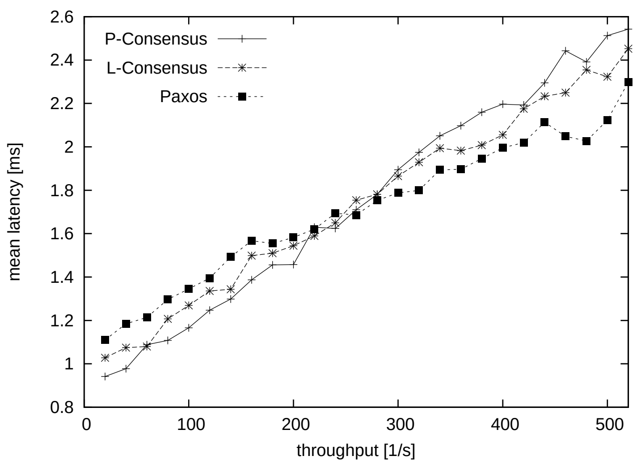

Figure 3: L-/P-Cons. ($n = 4$) vs. Paxos ($n = 3$)

a throughput of $80msg/s$ and they outperform WABCast for all throughputs higher than $100msg/s$. Figure 3 summarizes the comparison with Paxos. When collision predominate, the proposed protocols indeed have the same time complexity as Paxos. However, given their decentralized nature, our protocols need more messages. From a throughput of $300msg/s$ upwards, Paxos slightly outperforms both protocols. For lower throughputs, **L**-/**P**-Consensus perform better than Paxos.

9 Conclusion

One-step decision and zero-degradation express the ability to reach consenus in one and two communication steps respectively, and protocols that satisfy them are optimal in this respect. We investigated if these properties are inherently incompatible and showed that they cannot be both satisfied using the Ω failure detector. As shown in [11], any Ω based procol that decides in two communication steps in every well-behaved run is also zero-degrading. This implies that the failure detector employed by Fast Paxos [13] is strictly stronger than Ω. Subsequently, we proposed two approaches to circumvent the established impossiblity result. The first approach relaxes one-step decision to hold only in stable runs. The second approach assumes a strictly stronger failure detector. For each approach we developed a corresponding consensus protocol. While the proposed **L**-Consensus ensures one-step decision only in stable runs, the ability of **P**-Consensus to decide in one communication step is regardless of the failure detector output. To be able to test the efficiency of the proposed protocols we modified the atomic broadcast algorithm of [19] to use consensus. We compared the proposed consensus protocols with Paxos and WABcast both analytically and experimentally. The results of the experiments confirm the analytical evaluation establishing the efficiency of our proposed protocols.

Acknowledgments: We gratefully acknowledge the help and insights from Dr Falk Fraikin, the DEEDS group, the funding support from Microsoft Research via the European PhD Fellowship; and also from the EU DECOS and ReSIST projects .

References

[1] M. K. Aguilera *et al.* Failure detection and consensus in the crash-recovery model. *Dist. Computing*, vol. 13, 2, pp. 99-125, 2000.

[2] F. V. Brasileiro *et al.* Consensus in one communication step. *Proc. of PACT*, pp. 42-50, 2001.

[3] L. Camargos *et al.* Optimal and practical WAB-based consensus algorithms. *UNISI TR IC-05-07* Apr. 2005.

[4] T. D. Chandra *et al.* The weakest failure detector for solving consensus. *JACM*, vol. 43, 3, pp. 685-722, 1996.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, vol. 43, 2, pp. 225-267, 1996.

[6] F. Chu. Reducing Ω to $\Diamond\mathcal{W}$. *Inf. Processing Letters*, vol. 67, 6, pp. 289-293, 1998.

[7] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *Proc. FTCS*, pp. 140-149, 1998.

[8] C. Dwork *et al.* Consensus in the presence of partial synchrony. *JACM*, vol. 35, 2, pp. 288-323, 1988.

[9] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. *Proc. EDCC-4*, pp. 191-208, 2002.

[10] M. J. Fischer *et al.* Impossibility of distributed consensus with one faulty process. *JACM*, vol. 32, 2, pp. 374-382, 1985.

[11] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Trans. Computers*, vol. 53, 12, pp. 453-466, 2004.

[12] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *ACM SIGACT News*, Online Vol. 32, 2001.

[13] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, vol. 16, 2, pp. 133-169, 1998.

[14] L. Lamport. Lower bounds for asynchronous consensus. *Future Directions in Dist. Computing*, 2004.

[15] L. Lamport. Fast Paxos. *MSR TR 2005-112*, July 2005.

[16] N. A. Lynch. *Distributed Algorithms. Morgan Kaufmann Publishers*, 1996.

[17] A. Mostéfaoui and M. Raynal. Low cost consensus-based atomic broadcast. *Proc. PRDC*, pp. 45-54, 2000.

[18] F. Pedone and A. Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. *Journal of Theoretical Computer Science*, vol. 291, 1, pp. 79-101, 2003.

[19] F. Pedone *et al.* Solving agreement problems with weak ordering oracles. *Proc. of EDCC*, pp. 44-61, 2002.

[20] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, vol. 22, 4, pp. 299-319, 1990.

[21] P. Urbán *et al.* Neko: A single environment to simulate and prototype distributed algorithms. *Proc. of Information Networking*, pp. 503-511, 2001.

[22] P. Dutta *et al.* The Overhead of Consensus Recovery. *IC TR 200456*, June 2004.