Optimal and Practical WAB-Based Consensus Algorithms*

Lasaro Camargos[1,2], Edmundo R. M. Madeira[1], and Fernando Pedone[2]

[1] State University of Campinas, Brazil
{lasaro,edmundo}@ic.unicamp.br
[2] University of Lugano, Switzerland
fernando.pedone@unisi.ch

Abstract. In this paper we introduce two new WAB-based consensus algorithms for the crash-recovery model. The first one, B*-Consensus, is resilient to up to $f < n/2$ permanent faults, and can solve consensus in three communication steps. R*-Consensus, our second algorithm, is $f < n/3$ resilient, and can solve consensus in two communication steps. These algorithms are optimal with respect to the time complexity versus resilience tradeoff. We compare our algorithms to other consensus algorithms in the crash-recovery model.

Keywords: Consensus, crash-recovery, weak ordering oracles, Paxos.

1 Introduction

The consensus problem is a fundamental building block in fault-tolerant distributed systems. In a seminal paper, Fischer, Lynch, and Patterson have shown that consensus cannot be deterministically solved in a completely asynchronous distributed system subject to process failures [1]. This result implies that any consensus algorithm requires extensions to the pure asynchronous model if at least one process may crash during the execution.

Motivated by this theoretical bound, several approaches have been proposed to solve consensus by strengthening the asynchronous. Dolev et al. [2] and Dwork et al. [3] studied the minimum synchronization requirements needed by consensus. In [4], Chandra and Toueg introduced the concept of unreliable failure detectors, oracles that provide possibly incorrect information about process failures. Unreliable failure detectors encapsulate the synchronous assumptions needed to solve consensus and provide abstract properties to processes. The authors classified failure detectors in eight classes and showed that $\diamond\mathcal{W}$ encapsulates the minimal assumptions needed to solve consensus [5]. Some proposals have also considered solving consensus using a leader election oracle Ω [6, 7]. Intuitively, a leader election oracle ensures that nonfaulty processes eventually agree on the identity of some nonfaulty process, the leader.

* The work presented in this paper has been partially funded by CNPq, Brazil (grant 141749/2003-2), and SNSF, Switzerland (project #200021-103556).

Another way to circumvent the consensus impossibility result is to use randomization. The algorithms presented in [8, 9] use a random number generator to guarantee that with probability one processes will reach a decision. Algorithms similar to those in [8, 9] were presented by Pedone et al. [10]. Instead of relying on randomization, however, progress is ensured using *weak ordering oracles*. Such oracles provide message ordering guarantees but, as unreliable failure detectors and Ω, they can make mistakes. More specifically, the algorithms in [10] use the *weak atomic broadcast (WAB)* oracle. WAB ensures that if processes keep exchanging broadcast messages, then some of these messages will be delivered in the same order by all nonfaulty processes. Weak ordering oracles are motivated by Ethernet broadcast, present in many clustered architectures.

Lower bounds on what consensus algorithms can achieve have been also considered in the literature. Lamport summarizes previous results (e.g., [11, 12]) and presents new ones in [13]. These bounds show a tradeoff between resilience and time complexity (i.e., the number of communication steps needed to solve consensus). Briefly, the following results are stated: (a) To ensure progress, at least a majority of processes needs to be nonfaulty. (b) To allow a decision to be reached in two communication steps when more than one process is allowed to propose, more than two-thirds of the processes should be nonfaulty.

Despite the great interest that consensus has attracted and the multitude of algorithms that have been proposed to solve it, most works have considered system models which are of more theoretical than practical interest. This is mainly reflected in two aspects: the failure behavior of processes and the reliability of communication links. From a practical perspective, processes should be capable of re-integrating the system after a crash. Moreover, algorithms capable of tolerating message losses can make better use of highly-efficient communication means (e.g., UDP messages). We call such algorithms *practical*.

In this paper we introduce practical WAB-based consensus algorithms. Differently from those in [10], our protocols assume that processes can recover after failures and messages can be lost. The first one, B*-Consensus, is resilient to up to $f < n/2$ permanent failures; it solves consensus in three communication steps when the WAB oracle works. The second algorithm, R*-Consensus, is $f < n/3$ resilient and can solve consensus in two communication steps. Therefore, besides practical, our algorithms are also optimal regarding the time complexity versus resilience tradeoff.

The rest of the paper is organized as follows. We introduce our computational model and some definitions in Section 2. In Section 3 we present the B*-Consensus and the R*-Consensus algorithms. Correctness proofs for both algorithms can be found in [14]. In Section 4 we compare B*-Consensus and R*-Consensus to other practical consensus algorithms, and relate them to other works. Section 5 concludes the paper.

2 Model and definitions

2.1 Processes, communication, and failures

We consider an asynchronous system composed of a set $\Pi = \{p_1, \ldots, p_n\}$ of processes, $n \geq 3$. Processes communicate by message passing. Messages can be lost or duplicated but not corrupted. Processes can crash and recover an unlimited number of times but do not behave maliciously (i.e., no Byzantine failures). To ensure liveness we assume that eventually a subset of processes remains up forever. Such processes are called *stable*.

In the following sections we provide a definition of the consensus problem and then augment the asynchronous model with further assumptions to consensus solvable.

2.2 The consensus problem

Processes executing consensus can propose a value, interact to accept a single value, and learn the decision. Similarly to [13], we consider that these roles can be played independently by each process. Characterizing processes as *proposers*, *acceptors*, and *learners* allows us to simplify the algorithm's presentation. It also better models some real systems, e.g., it adequately matches a system where clients propose values to servers and then, without participating in the decision protocol themselves, learn the value accepted.

Using the decomposition of roles, consensus is defined as follows:

Nontriviality: only a proposed value may be learned.
Consistency: any two values that are learned must be equal.
Progress: for any proposer p and learner l, if p, l and $n - f$ acceptors are *stable*, and p proposes a value, then l must learn a value.

2.3 Weak Ordering Oracles

Weak ordering oracles provide message ordering guarantees [10]. A WAB is a weak ordering oracle defined by the primitives w-broadcast(k, m) and w-deliver(k, m), where $k \in \mathbb{N}$ defines a w-broadcast instance, and m is a message. The invocation of w-broadcast(k, m) broadcasts message m in instance k; w-deliver(k, m) w-delivers a message m w-broadcast in instance k. WAB satisfies the following property:

– If w-broadcast(k,–) is invoked in an infinite number of instances k, then (*Fairness*) for every instance k there is an instance $k' \geq k$ in which every stable process w-delivers a message and (*Spontaneous Order)* the first w-delivered message in instance k' is the same for every process that w-delivers a message in k'.

For example, consider an instance k in which processes p and q w-broadcast messages m_p and m_q respectively. If all non crashed processes in the system execute w-deliver$(k, _)$, and their first invocation of w-deliver returns the same message $m \in \{m_p, m_q\}$, then the property is satisfied in k. If, otherwise, some non crashed process does not execute w-deliver, or if one process w-delivers m_p while another w-delivers m_q, the property is not satisfied in i.

WABs are motivated by empirical observation of the behavior of IP-multicast in some local-area networks (e.g., Ethernet). In such environments, IP-multicast ensures that most broadcast messages are delivered in the same order to all network nodes.

3 B*-Consensus and R*-Consensus

In this section we introduce two WAB-based consensus algorithms for the crash-recovery model: B*-Consensus and R*-Consensus. These protocols were inspired by those in [10] but, differently from them, tolerate an unbounded number of failures and message losses without losing consistency. For example, any process can crash and recover an unbounded number of times. To ensure progress, however, a certain number of processes is required to be stable: B*-Consensus requires $f < n/2$ stable processes, and R*-Consensus requires $f < n/3$. By requiring more stable processes, R*-Consensus may reach a decision in two communication steps, while B*-Consensus needs at least three steps. Therefore, they have optimal time complexity [13]. To the best of our knowledge, these are the first WAB-based algorithms to consider crash-recovery failures and message loses.

As the two algorithms share some behavior, in the following sections we initially describe their commonalities and then describe their particularities.

3.1 General overview

R*-Consensus and B*-Consensus execute a sequence of rounds. In each round, proposers can propose a value, acceptors try to choose a value proposed in the round, and learners try to identify whether a decision has been made in the current round or if a new round must be started.

In a deciding round r, (i) a proposer w-broadcasts a value v, that is, it executes w-broacast(r, v); (ii) acceptors w-deliver some proposed value, possibly interact to accept a value, and notify the learners; and (iii) the learners, after gathering enough acceptance messages, learn that a value was decided and tell the application. The main difference between the algorithms lies in the meaning of *enough*; in order to decide with fewer messages, increasing the resilience from $f < n/3$ to $f < n/2$, acceptors in B*-Consensus must execute an extra communication round before accepting a value.

The algorithms are divided in blocks of statements, each one executed until completion and one at time. The *Initialization* block runs when the algorithm is started. If the process is recovering from a crash, the *Recovery* block is run,

instead. The other blocks have clauses triggered by message arrivals (receive and w-deliver), and only run after *Initialization* or *Recovery* have run.

In both algorithms, every process p keeps a variable r_p with the highest-numbered round in which p took part, and a variable $prop_p$ that either has the proposal for round r_p or $\perp$, meaning that any value can be proposed. Variables $prop_p$ and r_p are always logged together (see Algorithms 1 and 2), ensuring that processes replay rounds consistently, after recovering from a crash.

Skipping rounds. When a process p in round r_p sees a message sent in round $r_q > r_p$, p immediately jumps to r_q without performing rounds $r_p{+}1..r_q{-}1$. This allows processes that were down for a long time to rapidly catch up with the most advanced processes. Not every value is a valid proposal for every round: after deciding rounds, for example, only the decided value can be proposed. As only processes that finished round $r_q - 1$ initially know which values can be proposed in r_q, processes in earlier rounds must learn, maybe indirectly, which values are valid in r_q from processes in later rounds. This is accomplished by having each process' proposal attached to every message it sends (the last field of each message in the algorithm). The *Round Skipping Task*, presented in Algorithms 1 and 2, lines 8-16, runs on every message received/w-delivered before other clauses handle them. The algorithms in [15] can also skip rounds, but the procedure is more complicated than the one we present.

Proposers. Proposers are given a value by the application and try to pass it as the instance's decision. Due to message losses and process crashes, a consensus instance may not terminate in the first attempt, and may have to be retried. At any time, proposers can retry a consensus instance if they believe that the previous attempt has failed; consistency is kept even if the previous attempt is actually still running. To be able to learn that a round of the algorithm has terminated we assume that each proposer is also a learner. So, if a proposer does not learn the decision of the consensus it has initiated after some time, it re-starts its execution by proposing in its current round. Proposers execute lines 17-20 of the algorithms.

3.2 The B*-Consensus algorithm

Algorithm 1 presents the B*-Consensus algorithm.

Acceptors. In the B*-Consensus algorithm, every acceptor p will accept the first proposal w-delivered. That is, p takes this proposal as its first estimative ($est1_p$), and logs it together with the current round number (r_p) and a valid proposal. Then, p exchanges its estimative with other acceptors using CHECK messages, collecting $\lceil (n+1)/2 \rceil$ estimatives. p uses them as its second estimative ($est2_p$) if they are all equal, or $\top$, otherwise. p then logs $est2_p$ and r_p and sends them both to the learners in SECOND messages.

Learners. Once a learner has received $\lceil (n+1)/2 \rceil$ SECOND messages, it checks whether all carry the same estimative v. If that is the case, a decision has been reached and v is delivered to the application. Otherwise, p looks for at least one $v \neq \top$ in SECOND messages to be used as a proposition for the next round, so that any future rounds will only be able to decide v.

Algorithm 1 The B*-Consensus Algorithm

```
1: Initialization:
2: r_p ← 0
3: prop_p ← est1_p ← est2_p ← ⊥
4: Cset ← Sset ← ∅

5: Recovery:
6: retrieve(r_p, prop_p, est1_p, est2_p)
7: Cset ← Sset ← ∅

8: Round Skipping Task:
9: before executing receive(_, r_q, ...)
                or w-deliver(m, r_q)
10:   if r_p > r_q
11:     send (SKIP,r_p, prop_p) to q
12:   if r_p < r_q
13:     r_p ← r_q
14:     prop_p ← prop_q
15:     est1_p ← est2_p ← ⊥
16:     Cset ← Sset ← ∅

17: To propose value v_p do as follows:
18: if prop_p = ⊥
19:   prop_p ← v_p
20: w-broadcast (FIRST,r_p, prop_p)
                          to acceptors
21: Acceptors execute as follows:
22: upon w-deliver (FIRST, r_p, prop_q)
23:   if est1_p = ⊥
24:     est1_p ← prop_q
25:     log (est1_p, r_p, prop_p)
26:     send (CHECK, r_p, est1_p, prop_q)
                          to acceptors
27: upon receive (CHECK, r_p,est1_q,prop_q)
28:   Cset ← Cset ∪ {(CHECK, r_p, est1_q, prop_q)}
29:   if |Cset| = ⌈(n + 1)/2⌉
30:     if ∀(CHECK, r_p, est1_q, −) ∈ Cset :
                              est1_q = v
31:       est2_p ← v
32:     else
33:       est2_p ← ⊤
34:     log (est2_p, r_p, prop_p)
35:     send (SECOND, r_p, est2_p, prop_p)
                              to learners

36: Learners execute as follows:
37: upon receive (SECOND, r_p, est2_q, v_q)
38:   Sset ← Sset∪{(SECOND, r_p, est2_q, v_q)}
39:   if |Sset| = ⌈(n + 1)/2⌉
40:     if ∀(SECOND, r_p, est2_q, _) ∈ Sset :
                              est2_q = v ≠ ⊤
41:       decide v
42:     if ∃(SECOND, r_p, est2_q, _) ∈ Sset :
                              est2_q = v ≠ ⊤
43:       prop_p ← v
44:     r_p ← r_p + 1
45:     est1_p ← est2_p ← ⊥
46:     Cset ← Sset ← ∅
```

3.3 The R*-Consensus algorithm

Algorithm 2 presents the R*-Consensus algorithm.

Acceptors. In the R*-Consensus algorithm, as in B*-Consensus, every acceptor p accepts the first proposal it w-delivers, and logs it together with r_p and $prop_p$, so that these values will not be forgotten in case of crash. p then sends these values to all learners in SECOND messages.

Learners. Learners gather $\lceil (2n+1)/3 \rceil$ SECOND messages and check whether they contain the same estimative v. If that is the case, v is decided and delivered to the application. Otherwise, learners check if at least a majority of the estimatives are equal to v' in which case $prop_p$ is set to v', locking the value for future decisions. In any case, r_p is incremented.

Algorithm 2 The R*-Consensus Algorithm

```
1: Initialization:
2: r_p ← 0
3: prop_p ←, est1_p ← ⊥
4: Sset ← ∅

5: Recovery:
6: retrieve(r_p, prop_p, est1_p)
7: Sset ← ∅

8: Round Skipping Task:
9: before executing receive(_, r_q, ...)
                or w-deliver(m, r_q)
10:   if r_p > r_q
11:     send (SKIP,r_p, prop_p) to q
12:   if r_p < r_q
13:     r_p ← r_q
14:     prop_p ← prop_q
15:     est1_p ← ⊥
16:     Sset ← ∅

17: To propose value v_p do as follows:
18: if prop_p = ⊥
19:   prop_p ← v_p
20: w-broadcast (FIRST,prop_p)
                          to acceptors

21: Acceptors execute as follows:
22: upon deliver (FIRST,r_p, prop_q)
23:   if est1_p = ⊥
24:     est1_p ← prop_q
25:     log (est1_p, r_p, prop_p)
26:     send (SECOND, r_p, est1_p, prop_p)
                                  to learners

27: Learners execute as follows:
28: upon deliver (SECOND,r_p, est1_q, v_q)
29:   Sset ← Sset ∪ {(SECOND, r_p, est1_q, v_q)}
30:   if |Sset| = ⌈(2n + 1)/3⌉
31:     if ∀(SECOND, r_p, est1_q, _) ∈ Sset :
                                  est1_q = v
32:       decide v
33:     if ∃v_maj, for ⌈(n + 1)/2⌉ (SECOND, r_p, v, _)
                              ∈ Sset : v = v_maj
34:       prop_p ← v_maj
35:     else
36:       prop_p ← ⊥
37:     r_p ← r_p + 1
38:     est1_p ← ⊥
39:     Sset ← ∅
```

4 Related Work

WAB-based consensus algorithms were introduced in [10]. This work assumed crash-stop failures and reliable channels. Here we presented WAB-based consensus algorithms that allow processes to recover and messages to be lost.

The problem of consensus in the crash-recovery model was previously studied in [7, 15–19]. These approaches considered either a leader-election oracle or unreliable failure detectors (UFD) as extensions to the asynchronous model. In [15], Aguilera et al. showed that if the number of processes that never crash ("always-up processes") is bigger than the number of processes that eventually remain crashed or that crash and recovery infinitely many times, then consensus is solvable without stable storage; without this assumption stable storage is needed. As we do not bound the number of processes that are allowed to crash, our algorithms must use stable storage, although this is done sparingly. Differently from our approach, the algorithms in [16, 19] keep all their variables in stable storage and cannot be considered practical.

In [13] some lower bounds on how fast, in terms of communication steps, a consensus algorithm can be are given. Roughly, if any value proposed by two or more proposers can be decided within two communication steps, then no more than $f < n/3$ processes can be unstable; to be able to decide in three

communication steps, no more than $f < n/2$ processes can be unstable. Some algorithms found in the literature may decide in two communication steps and still be $f < n/2$ resilient. In these algorithms, however, only the value proposed by the coordinator[3] can be decided in two steps; deciding on a value proposed by other processes requires at least one message step more. As this extra step is important in several practical situations, e.g., when using consensus to implement atomic broadcast, in the following analysis we consider this extra step whenever it applies. Since WAB-based algorithms do not have the role of a coordinator, they do not suffer from this shortcoming.

The Paxos algorithm [7, 18] relies on an Ω leader election oracle to solve consensus. In its normal form, Paxos needs at least four (plus one) communication steps to decide on a value. By omitting the first phase of the algorithm, a simple optimization for good runs, two communication steps can be saved. Another variation of Paxos, Fast Paxos [20], eliminates the extra step by having any proposer proposing on behalf of the leader. In good runs, a decision can be reached in two communication steps. Since Paxos is $f < n/2$ resilient and Fast Paxos is $f < n/3$, they are optimal. Just like Paxos and Fast Paxos, the WAB-based algorithms we presented here are also optimal.

Hurfin et al. [17] presented an algorithm that has the same message pattern as Paxos in optimized mode, i.e., two (plus one) communication steps. Because it uses the rotating coordinator paradigm, the decision may be delayed when coordinators, elected deterministically, crash.

The algorithm relying on stable storage in [15] is $f < n/2$ resilient. In best-case runs, processes access stable storage twice in a round and reach decision within three (plus one) communication steps. B*-Consensus writes in disk twice in a round, while R*-Consensus writes only once. In Paxos, disk writes happen once per round in the optimized mode, and twice in the normal mode, that is, in each mode it has the same cost as one of our algorithms. Fast Paxos writes once per round, as does the algorithm in [17].

Table 1 summarizes consensus algorithms in terms of communication steps (i.e., expected latency), number of messages, their resilience, number of disk writes, and the oracle needed for termination. δ denotes the expected network delay assumed for the analysis of the algorithms. We consider both point-to-point and multicast communication, and assume that either one or the other is used at each configuration, but not both at the same time. Notice that we consider messages sent from a process to itself, as these messages also impose some processing cost at each machine.

From Table 1, the optimized version of Paxos takes the same number of communication steps as B*-Consensus but, due to its centralized nature, needs nearly half the messages. Two more communication steps are required when Paxos runs the first phase. Although the number of messages is half of B*-Consensus with point-to-point communication, it becomes almost the same when broadcast is available. Moreover, if the proposer is the current leader, then one communication step and one message can be saved in Paxos. When compared to R*-

[3] Leader and initiator are also names commonly used.

Protocol	**Expected**	**Number of Messages**		**Resilience**	**Oracle**	**Disk**
	Latency	**Unicast**	**Broadcast**			**Writes**
B*-Consensus	3δ	$2n^2 + n$	$2n + 1$	$f < n/2$	WAB	2
R*-Consensus	2δ	$n^2 + n$	$n + 1$	$f < n/3$	WAB	1
Fast Paxos	2δ	$n^2 + n$	$n + 1$	$f < n/3$	Ω	1
Paxos (optimized)	3δ	$n^2 + n + 1$	$n + 2$	$f < n/2$	Ω	1
Paxos (normal)	5δ	$n^2 + 3n + 1$	$2n + 3$	$f < n/2$	Ω	2
Aguilera et. al	4δ	$3n + 1$	$n + 3$	$f < n/2$	UFD	2
Hurfin et. al	3δ	$n^2 + n + 1$	$n + 2$	$f < n/2$	UFD	1

Table 1. Consensus algorithms in the crash-recovery model

Consensus, the optimized version of Paxos uses the same number of messages, and trades one communication step for better resilience: $f < n/2$ instead of $n < n/3$. Finally, notice that Paxos always degenerate to the normal case after the first try to achieve consensus fails using the optmized version of the protocol. Fast Paxos equals R*-Consensus in all criteria but the oracle. Aguilera et al.'s algorithm has the same resilience, latency and number of disk writes as B*-Consensus, but is more efficient in terms of messages. Hurfin et al.'s algorithm is just as efficient as R*-Consensus, but has better resilience. If it is important for more than one proposer to be able to have its proposal decided, then the WAB-based consensus algorithms become one communication step more efficient. Moreover, in our analysis we do not count the messages needed to implement the Ω and UFD abstractions.

5 Conclusions

In this paper we introduced B*-Consensus and R*-Consensus, two WAB-based algorithms that assume the crash-recovery model and tolerate message losses. Both algorithms can cope with any number of process failures without violating safety. B*-Consensus takes three communication steps to reach a decision and requires a majority of stable processes to ensure progress. R*-Consensus can decide in two communication steps, but requires more than two thirds of stable processes for progress. Both algorithms are optimal in terms of communication steps for the resilience they provide.

We compared our algorithms to other well-known consensus algorithms in the crash-recovery model. Due to their decentralized fashion, when using these protocols, any proposer may have its value decided within the minimal latency. This comes at the cost of resilience or extra messages. In the case of Fast Paxos, the only difference is the oracle used, Ω, and the number messages needed to implement it.

References

1. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2) (1985) 374–382
2. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM (JACM) **34**(1) (1987) 77–97
3. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2) (1988) 288–323
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM (JACM) **43**(2) (1996) 225–267
5. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM (JACM) **43**(4) (1996) 685–722
6. Dutta, P., Guerraoui, R.: Fast indulgent consensus with zero degradation. Lecture Notes in Computer Science **2485** (2002)
7. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS) **16**(2) (1998) 133–169
8. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing, ACM Press (1983) 27–30
9. Rabin, M.O.: Randomized byzantine generals. In: Proc. of the 24th Annu. IEEE Symp. on Foundations of Computer Science. (1983) 403–409
10. Pedone, F., Schiper, A., Urban, P., Cavin, D.: Solving agreement problems with weak ordering oracles. In: 4th European Dependable Computing Conference (EDCC-4), Toulouse, France (2002)
11. Charron-Bost, B., Schiper, A.: Uniform consensus is harder than consensus. J. Algorithms **51**(1) (2004) 15–37
12. Keidar, I., Rajsbaum, S.: On the cost of fault-tolerant consensus when there are no faults: preliminary version. SIGACT News **32**(2) (2001) 45–63
13. Lamport, L.: Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, Microsoft Research (2004)
14. Camargos, L., Pedone, F., Madeira, E.: Optimal and practical wab-based consensus algorithms. Technical Report IC-05-07, Institute of Computing, State University of Campinas, Campinas, Brazil (2005)
15. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. In: Proc. of the 12th International Symposium on Distributed Computing. (1998)
16. Dolev, D., Friedman, R., Keidar, I., Malkhi, D.: Failure detectors in omission failure environments. In: Symposium on Principles of Distributed Computing. (1997) 286
17. Hurfin, M., Mostefaoui, A., Raynal, M.: Consensus in asynchronous systems where processes can crash and recover. In: Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems, IEEE Comput., Soc, Los Alamitos, CA (1998) 280–286
18. Oki, B.M., Liskov, B.H.: Viewstamped replication: A new primary copy method to support highlyavailable distributed systems. In: PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, New York, NY, USA, ACM Press (1988) 8–17
19. Oliveira, R., Guerraoui, R., Schiper, A.: Consensus in the crash-recover model. Technical Report TR-97/239, EPFL – Départment d'Informatique, Lausanne, Switzerland (1997)
20. Lamport, L.: Lower bounds for asynchronous consensus. Technical Report MSR-TR-2005-112, Microsoft Research (2005)