Abstracting out Byzantine Behavior

Peter Druschel, Andreas Haeberlen, Petr Kouznetsov

Max Planck Institute for Software Systems
MPI-SWS, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
{druschel,ahae,pkouznet}@mpi-sws.mpg.de

Abstract. Many distributed systems are designed to tolerate the presence of *Byzantine* failures: an individual process may arbitrarily deviate from the algorithm assigned to it. Depending on the application requirements, systems enjoy various levels of fault-tolerance. Systems based on state machine replication are able to *mask* failures so that their effect is not visible by the application. In contrast, cooperative peer-to-peer systems can tolerate bounded deviant behavior to some extent and therefore do not require masking, as long as each faulty node is *exposed* eventually. Finding an abstract way to reason about the levels of fault-tolerance is thus of immanent importance.
In this paper, we discuss how the information of deviant behavior can be abstracted out in the form of a *Byzantine failure detector* (BFD). We formally define a BFD abstraction, and we discuss two ways of using the abstraction: (1) monitoring systems in order to retroactively detect Byzantine failures and (2) enforcing systems in order to boost their level of fault-tolerance. Interestingly, the BFD formalism allowed us to determine the relative hardness of implementing two popular abstractions in distributed computing: state machine replication and weak interactive consistency.

Keywords. Fault-tolerance, Byzantine failures, masking, detection, Byzantine failure detectors, total order broadcast, weak interactive consistency.

1 Introduction

A distributed system might exhibit complex and unpredictable behavior when some of its components are subject to failures. *Crash-stop failures* represent a “benign” class of failures: a crashed process simply prematurely halts all its activities. In an *asynchronous* system where no assumption about message delays and process relative speeds can hold, there is no way to reason about crash-stop failures, since it is impossible to distinguish a crashed process from a “slow” one in a finite execution. As a result, even if only one process is allowed to crash, most important agreement problems cannot be solved [1].

Failure detectors. Chandra and Toueg proposed to enrich the asynchronous system with modular devices, called *failure detectors*, that encapsulate the timing assumptions [2]. A failure detector is a distributed oracle that provides processes with some information about failures. At any point of an execution, this information can however be incomplete and unreliable. The output of a failure detector is specified through a set of abstract axiomatic properties, so that a failure detector can be implemented in any particular environment independently of an algorithm that uses the failure detector. Thus, algorithms using failure detectors can be designed and proved without looking at the actual implementations of these failure detectors.

From a more theoretical perspective, the failure detector abstraction allows us to determine precisely the exact timing assumptions necessary to solve a given problem in distribute computing. These exact timing assumptions can be expressed in the form of the weakest failure detector [3]. The weakest failure detector for a given problem captures the weakest systems model in which the problem can be solved, which in turn can be used for evaluating relative "hardness" of problems and classifying them. This approach brought a number of interesting insights on solvability of and relations between various fundamental problems in distributed computing, e.g., solving consensus [3] and non-blocking atomic commitment [4], emulating shared memory [5], and implementing a distributed lock [6].

Byzantine failures. In this paper, we focus on the most general class of failures: a *Byzantine* faulty process may deviate from its specification in an arbitrary way. In particular, instead of crashing, a Byzantine process can maliciously deviate from its specification, e.g., trying to confuse correct processes and bring the whole system in an inconsistent state. We address here the following question: would it make sense to extend the notion of a failure detector to general Byzantine failure model?

Since the definition of Byzantine failures involves the knowledge of the system specification, i.e., the automata associated with each process, in a natural definition of a Byzantine failure detector (BFD), the expected correct behavior of the distributed system should be taken as a parameter.[1] However, we would like to reach a proper separation of concerns, so that computations performed by the processes were distinct from the failure detection mechanism. We assume in this paper that information output by a BFD depends only on failures and does not reveal other aspects of the current computation.

Obviously, certain incorrect state transitions, such as application-specific events, cannot be monitored remotely. In this paper, we restrict our attention to a large class of *detectable* failures, i.e., incorrect state transitions that causally affect at least one correct process. This class includes, besides crash-stop failures, *muteness* failures, when a process stops sending certain algorithmic messages (while possibly sending other messages) [8,9,10], *ignorance* failures, when a process refuses to accept certain messages, and incorrect state transitions affecting correct processes.

[1] The impossibility of building black-box Byzantine failure detectors is discussed exhaustively in [7].

A BFD can thus be defined as a function that maps a Byzantine failure pattern (specifying where and when failures occur) to a set of allowable failure detector histories (specifying what information about failures is output at each process). We demonstrate two scenarios in which this abstraction can be of use. First, BFD can be used for monitoring a system in order to provide the application with hints about observed Byzantine failures. Second, BFDs can enforce systems with information about failures in order to boost their level of fault-tolerance.

Monitoring systems. Eventual detection of Byzantine behavior, while not suitable for masking Byzantine failures [11], can be used for building an accountability service. To illustrate this, we introduce the following *deviant peer* Byzantine failure detector $\diamond\mathcal{B}$. At every process i, $\diamond\mathcal{B}$ outputs, for every other process j, indicators of whether j is (from i's perspective) *trusted*, *suspected*, and *exposed*. Informally, $\diamond\mathcal{B}$ guarantees that eventually, every detectably faulty process will be either forever suspected or forever exposed by every correct process, and no correct process is exposed or forever suspected by any other correct process.

The possibility of retroactively but reliably exposing detectable Byzantine behavior creates a disincentive for malicious processes to exhibit malicious behavior and imposes a time bound on the effect of such behavior. We show that, maybe surprisingly, such a service can be implemented in the *asynchronous* authenticated system.

Boosting fault-tolerance and comparing problems. We discuss the use of Byzantine failure detectors for improving fault-tolerance of distributed systems in the context of state machine replication systems [12,13,11]. In these systems, requests of clients are processed by a collection of servers in a total order so that, from a correct client's point of view, the execution is indistinguishable from an execution in which all requests are processed by a single correct server. Ideally, a replicated service should tolerate any number of faulty clients and a bounded number of faulty servers.

Recently, Doudou *et al.* [14] proposed an elegant modularization of a replicated service based on a *reliable broadcast* and a *weak interactive consistency* modules. Informally, reliable broadcast is a communication primitive that ensures that the correct processes eventually agree on the set of delivered messages. Weak interactive consistency is a form of agreement which makes sure that the correct processes eventually agree on a set of values which contains at least one value proposed by a correct process. Reliable broadcast can be implemented in any system in which correct processes are able to reliably communicate. Weak interactive consistency is in turn implemented, under the assumption that less than one third of processes can be faulty, using a *muteness* failure detector $\diamond\mathcal{S}^B$. $\diamond\mathcal{S}^B$ outputs a list of processes suspected to be *mute*, i.e., to prematurely stop sending algorithm messages, and which guarantees that there is a time after which (1) every mute process is forever suspected and (2) *some* correct process is never suspected by every correct process. Thus, the liveness properties of replication systems can be naturally achieved using a Byzantine failure detector.

An interesting question arises: is weak interactive consistency *necessary* to implement a replicated service? In other words, can WIC be implemented in any model in which a replicated service can be implemented?

We use our BFD framework to show that the answer is “no”: we present a Byzantine failure detector that can be used to implement a replicated service but is too weak to implement WIC. This implies that WIC is in a strict sense harder to implement than state machine replication. Interestingly, a similar modularization of state machine replication for the crash-stop failure model into reliable broadcast and consensus is optimal: consensus can be shown to be necessary to implement a replicated service [15].

Contributions. To summarize, the main contribution of this paper is a generic framework that allows us to specify a wide range of Byzantine failure detectors. We illustrate the usability of this framework by presenting an accountability service that monitors systems and retroactively detects failures. Further, we show how the BFD abstraction can be used to measure the relative hardness of solving Byzantine fault-tolerant problems. More precisely, the BFD framework allows us to show that weak interactive consistency of [14] is strictly harder to implement than a replicated service [12,13,11].

Roadmap. The paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we formulate our system model and introduce the notion of detectable failures. In Section 4, we introduce the BFD abstraction. In Section 5, we discuss the use of BFDs for monitoring systems, and in Section 6, we discuss how the BFD framework can be used for determining relative hardness of problems in distributed computing. Section 7 concludes the paper.

2 Related Work

There is a large body of literature related to various ways of dealing with Byzantine failures in distributed systems. The study of distributed systems that are tolerant to arbitrary deviations of individual processes was pioneered in 1980 by Pease *et al.* who introduced the Byzantine consensus problem [16].

Malkhi and Reiter [10] were the first to consider the notion of failure detectors in the Byzantine failure model. They have introduced the $\diamond\mathcal{S}(bz)$ failure detector in a model where processes communicate through a *Reliable Broadcast* primitive with additional causal order delivery guarantees. They proposed a solution for the binary consensus problem using $\diamond\mathcal{S}(bz)$, where only deviant behavior that prevents processes from reaching consensus is detected. In a more general way, Doudou and Schiper [8] and Doudou et al. [9] have introduced, in the context of the consensus problem, *muteness* failure detectors that deal with *mute* processes, which prematurely stop sending algorithm messages. Friedman *et al.* [17] have explored time-efficient consensus algorithms based on oracles detecting mute processes. Kihlstrom et al. [18] have taken a more general approach and have introduced several classes of failure detectors that expose *detectable* Byzantine failures. They have also presented a consensus algorithm using one of these failure detectors. A statistical evaluation of the number of Byzantine processes in quorum systems was proposed by Alvisi *et al.* in [19].

To our knowledge, however, our paper is first attempt to formalize the Byzantine Failure Detector abstraction in the most general way, detached from the problem space and specific assumptions about the underlying failure model.

3 Model

We consider a set Π of *nodes*. Every node i is modeled as a state machine $A_i = (S_i, I_i, O_i, \Delta_i)$, where S_i is a set of states i can take, I_i is a set of inputs i can accept, O_i is a set of outputs i can produce, and Δ_i is a relation that maps every state and input to a set of possible states and outputs. Without loss of generality, assume that every state machine is initially in a predefined state $\perp$.

Nodes communicate with each other through message passing. We assume that messages are uniquely identified. For a message m, let $\mathit{sender}(m)$ and $\mathit{receiver}(m)$ denote the sender and the receiver of m, respectively. We assume that every node digitally signs its messages (e.g. with [20]), and that signatures cannot be forged.

An *event* is either $\mathit{send}_i(m) \in O_i$, where $i = \mathit{sender}(m)$, or $\mathit{receive}_j(m) \in I_j$, where $j = \mathit{receiver}(m)$, or an application-specific input or output of the state machine. An *execution* E is a sequence of events such that in E, each m is sent and received at most once, and each $\mathit{receive}_i(m)$ is preceded by the corresponding $\mathit{send}_j(m)$. A *local execution* of a node i, denoted $E|i$, is a subsequence of E that consists of all events associated with i in E. We say that a node i is *correct* in E if $E|i$ *conforms* to A_i, i.e., if the sequence of outputs produced in E is legal, given A_i and the sequence of inputs in E. Otherwise we say that i is *faulty* in E. Every event in an execution is associated with a unique time at which the event took place.

For the moment, we do not put any restrictions on local processing time and communication delays. We assume, however, that channels are *reliable*, i.e. that every message sent from a correct node to a correct node is eventually received.

Clearly, not every type of faulty behavior can be detected by a correct node. For example, a faulty node might disclose its deviant behavior only to its application or to other faulty nodes. Therefore, we introduce the notions of *detectably faulty* and *detectably deaf* nodes. Informally, a node i is detectably faulty if the behavior it exposes to correct nodes could not be observed if i were correct, and detectably deaf if i ignores a message sent to i by a correct node.

Formally, we define a *history* of a node i as a sequence of events. A history h of a node i is *valid* if it conforms to A_i, i.e. if, given the sequence of incoming messages and application-specific inputs in h, A_i could have produced the sequence of outgoing messages and application-specific outputs in h. A pair (h_1, h_2) of histories of i is *consistent* if h_1 is a prefix of h_2, or vice versa. If i is a correct node, one trivial example of a valid history is $E|i$.

Let $\mathcal{M}(E)$ denote the set of messages received by the nodes in an execution E. We assume that there exists a *history map* φ that associates every message $m \in \mathcal{M}(E)$ with a history of *sender*(m). We assume that for a correct node, $\varphi(m)$ is the prefix of the local execution $E|$*sender*(m) up to and including *send*(m). Thus, for any message m sent by a correct node, $\varphi(m)$ is valid, and for every pair of messages m and m' sent by a correct node, $\varphi(m)$ and $\varphi(m')$ are consistent.

We say that a message m is *observable in* E if there exists a correct node i and a sequence of messages $m_1, \ldots, m_k$ such that

(i) $m_1 = m$,
(ii) *receive*(m_k) belongs to $E|i$,
(iii) for all $j = 2, \ldots, k$: *receive*(m_{j-1}) belongs to $\varphi(m_j)$.

We say that a node i is *detectably malicious* at time t in an execution E if there exists a message m that was sent by i at time $t' < t$, is observable in E, and satisfies one of the following properties:

(1) $\varphi(m)$ is not valid (for i)
(2) There exists a message m' that was also sent by i and is observable in E, such that $\varphi(m)$ is inconsistent with $\varphi(m')$

Let E be any infinite execution. We say that a node i is *detectably deaf* at time t in E if i is not detectably malicious in E and there exists a message m sent to i at time $t' < t$ by a correct node, such that, for all m' sent by i and received by a correct node, *receive*$_i(m)$ does not appear in $\varphi(m')$. We say that a node i is *mute* at time t in E if E has a suffix E' in which no correct process receives a message from i. A process is called *detectably faulty* at time t if it is detectably malicious, deaf, or mute at time t. Figure 1 depicts the relationships between these classes of failures. Note that crash-stop faulty processes are both mute and deaf.

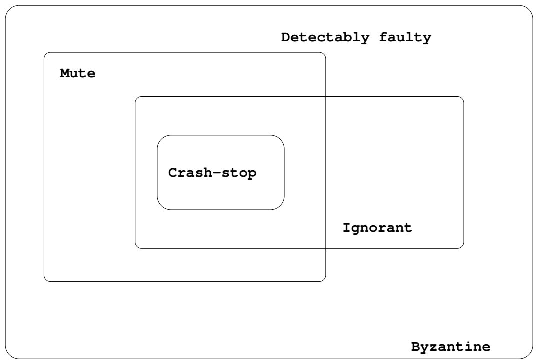

Fig. 1. Classes of failures

4 The Byzantine Failure Detector Abstraction

A *Byzantine Failure Detector (BFD)* is a distributed oracle: every process i is equipped with its own *BFD module*, which gives i some hints about deviant behavior of other processes. As an input, every BFD module takes the specification of the expected behavior of the underlying distributed system described in terms of state machines $\{A_j | j \in \Pi\}$. Moreover, the BFD module at process i is given the power to keep track of all inputs and outputs of A_i. Using this information, the BFD module provides i with *some* indications about who follows its specification and who does not. The information can be used in two ways (Figure 2). First, a BFD module can monitor other processes a provide the application level with indications about the suspected deviant behavior. Second,

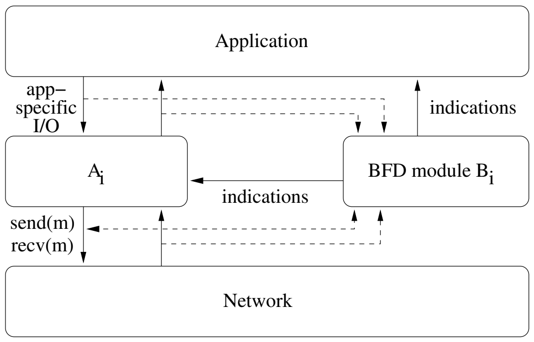

Fig. 2. BFD interface at process i

We do not restrict a priori the range of the BFD output. However, we require that only information based on failures can be output. In other words, a failure detector is not allowed to reveal any information about computations performed by correct processes.

5 Monitoring Systems with BFDs

We illustrate the use of BFDs for efficiently detecting Byzantine behavior by defining the following *deviant peer* BFD, denoted by $\Diamond\mathcal{B}$. This BFD provides the application level of every process i with indications $\mathit{trusted}_j$, $\mathit{suspected}_j$, and $\mathit{exposed}_j$ (for every process $j \neq i$). Intuitively, the $\mathit{suspected}_j$ indication suggests that j is ignoring certain inputs from the system, e.g., by refusing to accept a service request from a correct process (we say j is *suspected by* i). The $\mathit{exposed}_j$ indication means that i has a *proof* that j deviated from the specification of its state machine A_j (we say j is *exposed by* i). Finally, $\mathit{trusted}_j$ indicates that none of the aforementioned behaviors is observed by i (we say j is *trusted by* i). In every execution, $\Diamond\mathcal{B}$ ensures the following properties:

- Completeness
  1. (1) Eventually, every detectably deaf process is suspected by every correct process.
  2. (2) Eventually, every detectably faulty process is exposed by every correct process.
- Accuracy
  1. (1) No correct process is forever suspected by any correct process.
  2. (2) No correct process is ever exposed by a correct process.

Note that $\diamond\mathcal{B}$ is similar to the *eventually perfect* failure detector of [2] that outputs a set of suspected processes so that eventually exactly the set of *detectably deaf* processes is forever output at every correct process. However, $\diamond\mathcal{B}$ and $\diamond\mathcal{P}$ are, in a strict sense, incomparable. Indeed, $\diamond\mathcal{P}$ does not produce any information about detectably faulty processes. On the other hand, $\diamond\mathcal{B}$ does not guarantee that a correct process is always trusted by a correct process (it can jump from *trusted* to *suspected* and back). In fact, $\diamond\mathcal{B}$ can be implemented in a purely asynchronous system, which is not the case with $\diamond\mathcal{P}$.

In this , each message includes the full history of its sender and is broadcast to every node in the system. As soon as a process receives an invalid history or a pair of contradicting history it exposes the process and broadcasts the invalid histories as a proof. A process i is suspected as long as some correct process j sent a message m to i and no history from j that includes $receive_i(m)$ is received by any correct process.

Of course, this implementation is not very practical: it has $O(n^2)$ message complexity and it assumes that the full history is included in each message. A more realistic implementation of a slightly weaker monitoring service can be found in [21].

6 Boosting Fault-Tolerance with BFDs

In this section, we focus on algorithms for Byzantine fault-tolerant state machine replication. Formally, a replicated service is specified as the *total order broadcast* abstraction. Total order broadcast exports two primitives to-broadcast() and to-deliver() and satisfies the following properties:

validity: if a correct process i to-broadcasts a message m, then i eventually to-delivers m;

agreement: if a process to-delivers a message m, then every correct process eventually to-delivers m;

integrity: each process to-delivers every message at most once, and only if the message was previously to-broadcast;

total-order: if a process i to-delivers a message m before having to-delivered a message m', then no process j can to-deliver m' without having to-delivered m first.

Doudou *et al.* proposed in [14] an elegant decomposition of Byzantine fault-tolerant total order broadcast. They introduced the Weak Interactive Consistency (WIC) abstraction and presented an algorithm that implements total order broadcast using multiple instances of WIC and the reliable broadcast primitive.

In WIC, every correct process *proposes* a value and eventually the correct processes decide on the same set of proposed values that includes *at least one value proposed by a correct process.*

It is shown in [14] that the WIC abstraction is *sufficient* to implement total order broadcast. But is it also *necessary*? Can WIC be implemented in any model in which total order broadcast can be implemented?

We show that this is not the case by presenting a Byzantine failure detector that can be used to implement total order broadcast but is too weak to implement WIC. This failure detector is the composition $(\diamond\mathcal{S}^B, \Sigma^B)$. $\diamond\mathcal{S}^B$ outputs a list of processes suspected to be mute and guarantees that (1) eventually every mute process is forever suspected by every correct process, and (2) there is a time after which *some* correct process is never suspected by any correct process. Σ^B outputs a set of processes, called *quorum* such that (1) every two quorums output at any correct processes at any times share at least one correct process, and (2) eventually quorums output at every correct process include only correct processes.

Theorem 1. *No algorithm implements weak interactive consistency using $\diamond\mathcal{S}^B$ and Σ^B.*

Proof sketch. Assume by contradiction that an algorithm A implements WIC using $(\diamond\mathcal{S}^B, \Sigma^B)$. Let Q be a proper subset of processes and consider an execution E of A in which only processes in Q are correct. Assume that $(\diamond\mathcal{S}^B, \Sigma^B)$ outputs (j, Q) at every process in Q where $j \in Q$. Since E solves WIC, E has a prefix, denoted $\bar{E}$, in which every process in Q decides on a set of values S proposed by processes in Q. $\bar{E}$ can also be a prefix of an execution E' in which some process $i \notin Q$ is also correct. Clearly, E' has a prefix, denoted $\bar{E}'$, in which process i decides on S. But $\bar{E}'$ can also be a prefix of an execution E'' in which i is correct and all processes in Q are faulty. Thus, in E'', i decides on a set of values that includes no value proposed by a correct process — a contradiction. □

Theorem 2. *There is an algorithm that implements total order broadcast using $\diamond\mathcal{S}^B$ and Σ^B.*

Proof sketch. The following simple modification of the replication algorithm of Castro and Liskov [11] solves the problem.

(1) The safety properties of the replication algorithm of [11] are guaranteed by the *certification* and the *conservative communication* mechanisms, and the assumption that the number of processes n is greater than $3f$, where f is the number of processes that are allowed to fail in the same execution. The

certification is implemented as follows: a process that wants to certify a value sends the value to everybody and waits until the value is acknowledged by at least $f+1$ processes. As a result, by attaching the set of signed acknowledgements to the value, the process has a proof that the value is acknowledged by at least one correct process. To send a value in a conservative manner, a process sends the value and waits until at least $2f+1$ values acknowledge the value.
We replace every statement a process i broadcasts a value and waits for acknowledgements from $f+1$ or $2f+1$ processes with the following statement:

```
**repeat**
  $Q \leftarrow \Sigma^B$
**until** $Q \subseteq$ the set of processes from which acknowledgements are received
```

The cycle above is clearly non-blocking: Σ^B guarantees to eventually include only correct processes.
The set of values received from the quorum Q is attached to the values signed and sent by i in the same way as the set of $f+1$ and $2f+1$ values in the original algorithm of [11].

(2) To achieve liveness, in the algorithm of [11], processes maintain a coordinator (primary replica) as long as the coordinator is available. If the coordinator becomes unavailable (timed-out), processes exponentially increase the timeout and move to the next *view* by electing a new coordinator.in a round-robin fashion.
This mechanism can be naturally expressed using $\diamond\mathcal{S}^B$. Roughly speaking, a process moves to the next view as soon as the coordinator of the current view is suspected by $\diamond\mathcal{S}^B$. $\diamond\mathcal{S}^B$ guarantees that every coordinator which stops sending algorithm messages will eventually be suspected by every correct process. Thus no process can be blocked forever waiting for a faulty coordinator to reply. On the other hand, $\diamond\mathcal{S}^B$ guarantees that eventually some correct process is never suspected. Thus, eventually correct processes reach a view in which the coordinator is correct and never suspected.

Thus, safety and liveness of total order broadcast are preserved. □

Theorems 1 and 2 imply that WIC is in a strict sense harder to implement than total order broadcast.

7 Concluding Remarks

Handling Byzantine failures in an abstract and modular way is traditionally argued to be difficult. In this paper, we proposed a framework that formalizes the concept of Byzantine failure detectors in a generic manner. This allowed us to specify and build a simple but powerful accountability service that retroactively exposes any detectable Byzantine behavior. Then we use the framework to relate the problems of total order broadcast and weak interactive consistency (WIC) [14]. We presented a Byzantine failure detector that is strong enough to be used to implement total order broadcast but cannot be used to implement WIC. Combined with the construction presented in [14], this result implies that WIC is in a strict sense, harder than total order broadcast. We actually conjecture that Byzantine fault-tolerant total order broadcast cannot be *optimally* modularized using one-shot agreement tasks. This would contrast the crash-stop failure model in which such a modularization is feasible [15].

We believe that this paper opens several interesting research avenues. First of all, it would be interesting to understand whether we could define the notion of the *weakest* BFD to solve a given problem. This seems non-trivial since BFDs are defined with respect to algorithms that might use them, which makes them difficult to compare. However, as we show in this paper BFDs *can* be used to measure the respective hardness to solve problems.

In this paper, we assumed that BFDs do not reveal any information about correct state transitions. For some classes of systems, it might be convenient to remotely detect *some* correct state transitions, e.g., transitions to the next round in a round-based system. In the crash-stop failure model, this approach was elaborated in the round-by-round failure detector framework [22]. It would be interesting to analyze the implications of this approach in the Byzantine failure model.

References

1. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32(3)** (1985) 374–382
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43(2)** (1996) 225–267
3. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43(4)** (1996) 685–722
4. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., Toueg, S.: The weakest failure detectors to solve certain fundamental problems in distributed computing. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC). (2004) 338–346
5. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Shared memory vs message passing. Technical Report IC/2003/77, EPFL (2003) Available at http://icwww.epfl.ch/publications/.
6. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Dustributed Computing (JPDC) **65** (2005) 492–505
7. Doudou, A., Garbinato, B., Guerraoui, R.: Encapsulating failure detection: From crash to Byzantine failures. In: Ada-Europe. (2002) 24–50
8. Doudou, A., Schiper, A.: Muteness detectors for consensus with Byzantine processes. In: PODC. (1998) 315
9. Doudou, A., Garbinato, B., Guerraoui, R., Schiper, A.: Muteness failure detectors: Specification and implementation. In: EDCC. (1999) 71–87
10. Malkhi, D., Reiter, M.K.: Unreliable intrusion detection in distributed computations. In: CSFW. (1997) 116–125
11. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of OSDI'99, New Orleans, LA (1999) 173–186

12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21** (1978) 558–565
13. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22** (1990) 299–319
14. Doudou, A., Garbinato, B., Guerraoui, R.: Tolerating Arbitrary Failures with State Machine Replication. In: Dependable Computing Systems. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc (2005)
15. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcast and related problems. Technical report, Cornell University, Computer Science (1994)
16. Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27** (1980) 228–234
17. Friedman, R., Mostéfaoui, A., Raynal, M.: Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. IEEE Trans. Dependable Sec. Comput. **2** (2005) 46–56
18. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine fault detectors for solving consensus. Comput. J. **46** (2003) 16–35
19. Alvisi, L., Malkhi, D., Pierce, E.T., Reiter, M.K.: Fault detection for byzantine quorum systems. IEEE Trans. Parallel Distrib. Syst. **12** (2001) 996–1007
20. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21** (1978) 120–126
21. Haeberlen, A., Kouznetsov, P., Druschel, P.: PeerReview: Detecting faulty behavior in distributed systems. Technical Report Max Planck Institute for Software Systems 2006-1 (2006)
22. Gafni, E.: Round-by-round fault detectors : Unifying synchrony and asynchrony. (1998) 143–152