# The Fault Detection Problem

Andreas Haeberlen[1] and Petr Kuznetsov[2]

[1] Max Planck Institute for Software Systems (MPI-SWS)
[2] TU Berlin / Deutsche Telekom Laboratories

**Abstract.** One of the most important challenges in distributed computing is ensuring that services are correct and available despite faults. Recently it has been argued that fault detection can be factored out from computation, and that a generic fault detection service can be a useful abstraction for building distributed systems. However, while fault detection has been extensively studied for crash faults, little is known about detecting more general kinds of faults.
This paper explores the power and the inherent costs of generic fault detection in a distributed system. We propose a formal framework that allows us to partition the set of all faults that can possibly occur in a distributed computation into several *fault classes*. Then we formulate the *fault detection problem* for a given fault class, and we show that this problem can be solved for only two specific fault classes, namely *omission faults* and *commission faults*. Finally, we derive tight lower bounds on the cost of solving the problem for these two classes in asynchronous message-passing systems.

**Keywords:** Fault classes, fault detection problem, message complexity, lower bounds

## 1 Introduction

Handling faults is a key challenge in building reliable distributed systems. There are two main approaches to this problem: *Fault masking* aims to hide the symptoms of a limited number of faults, so that users can be provided with correct service in the presence of faults [4, 14], whereas *fault detection* aims at identifying the faulty components, so that they can be isolated and repaired [7, 10]. These approaches are largely complementary. In this paper, we focus on fault detection.

Fault detection has been extensively studied in the context of “benign” crash faults, where it is assumed that a faulty component simply stops taking steps of its algorithm [5, 6]. However, this assumption does not always hold in practice; in fact, recent studies have shown that general faults (also known as Byzantine faults [15]) can have a considerable impact on practical systems [17]. Thus, it would be useful to apply fault detection to a wider class of faults. So far, very little is known about this topic; there is a paper by Kihlstrom et al. [12] that discusses Byzantine fault detectors for consensus and broadcast protocols, and there are several algorithms for detecting certain types of non-crash faults, such

as PeerReview [10] and SUNDR [16]. However, many open questions remain; for example, we still lack a formal characterization of the types of non-crash faults that can be detected in general, and nothing is known about inherent costs of detection.

This paper is a first step towards a better understanding of general fault detection. We propose a formal model that allows us to formulate the *fault detection problem* for arbitrary faults, including non-crash faults. We introduce the notion of a *fault class* that captures a set of *faults*, i.e., deviations of system components from their expected behavior. Solving the fault detection problem for a fault class $F$ means finding a transformation $\tau_F$ that, given any algorithm $A$, constructs an algorithm $\bar{A}$ (called an *extension* of $A$) that works exactly like $A$ but does some additional work to identify and expose faulty nodes. Whenever a fault instance from the class $F$ appears, $\bar{A}$ must expose at least one faulty suspect (completeness), it must not expose any correct nodes infinitely long (accuracy), and, optionally, it may ensure that all correct nodes expose the same faulty suspects (agreement).

Though quite weak, our definition of the fault detection problem still allows us to answer two specific questions: Which faults can be detected, and how much extra work from does fault detection require from the extension? To answer the first question, we show that the set of all fault instances can be divided into four non-overlapping classes, and that the fault detection problem can be solved for exactly two of them, which we call *commission faults* and *omission faults*. Intuitively, a commission fault exists when a node sends messages a correct node would not send, whereas an omission fault exists when a node does *not* send messages a correct node *would* send.

To answer the second question, we study the *message complexity* of the fault detection problem, that is, the ratio between the number of messages sent by the most efficient extension and the number of messages sent by the original algorithm. We derive tight lower bounds on the message complexity for commission and omission faults, with and without agreement. Our results show that a) the message complexity for omission faults is higher than that for commission faults, and that b) the message complexity is (optimally) linear in the number of nodes in the system, except when agreement is required for omission faults, in which case it is quadratic in the number of nodes.

In summary, this paper makes the following four contributions: (1) a formal model of a distributed system in which various kinds of faults can be selectively analyzed, (2) a statement of the fault detection problem for arbitrary faults, (3) a complete classification of all possible faults, including a precise characterization of the set of faults for which the fault detection problem can be solved, and (4) tight lower bounds on the message complexity of the fault detection problem. Viewed collectively, our results constitute a first step toward understanding the power and the inherent costs of fault detection in a distributed system.

The rest of this paper is organized as follows: We begin by introducing our system model in Section 2 and then formally state the fault detection problem in Section 3. In Section 4, we present our classification of faults, and we show for

which classes the fault detection problem can be solved. In Section 5, we derive tight bounds on the message complexity, and we conclude by discussing related work in Section 6 and future work in Section 7. Omitted proofs can be found in the full version of this paper, which is available as a technical report [9].

## 2 Preliminaries

### 2.1 System model

Let $N$ be a set of *nodes*. Each node has a terminal[3] and a network interface. It can communicate with the other nodes by sending and receiving messages over the network, and it can send outputs to, and receive inputs from, its local terminal. We assume that processing times are negligible; when a node receives an input, it can produce a response immediately.

Each message $m$ has a unique *source* $src(m) \in N$ and a unique *destination* $dest(m) \in N$. We assume that messages are authenticated; that is, each node $i$ can initially create only messages $m$ with $src(m) = i$, although it can delegate this capability to other nodes (e.g., by revealing its key material). Nodes can also forward messages to other nodes and include messages in other messages they send, and we assume that a forwarded or included message can still be authenticated.

A computation unfolds in discrete *events*. An event is a tuple $(i, I, O)$, where $i \in N$ is a node on which the event occurs, $I$ is a set of inputs (terminal inputs or messages) that $i$ receives in the event, and $O$ is a set of outputs (terminal outputs or messages) that $i$ produces in the event. An *execution* $e$ is a sequence of events $(i_1, I_1, O_1), (i_2, I_2, O_2), \ldots$. We write $e|_S$ for the subsequence of $e$ that contains the events with $i_k \in S$; for $i \in N$, we abbreviate $e_{\{i\}}$ as $e|_i$. When a finite execution $e$ is a prefix of another execution $e'$, we write $e \lfloor e'$. Finally, we write $|e|$ to denote the number of unique messages that are sent in $e$.

A system is modeled as a set of executions. In this paper, we assume that the network is reliable, that is, a) a message is only received if it has previously been sent at least once, and b) a message that is sent is eventually received at least once. Formally, we assume that, for every execution $e$ of the system and every message $m$:

$$m \in I_k \Rightarrow [i_k = dest(m) \wedge \exists l < k : (i_l = src(m) \wedge m \in O_l)]$$

$$(m \in O_k \wedge src(m) = i_k) \Rightarrow [\exists l : i_l = dest(m) \wedge m \in I_l]$$

An *open execution* is an execution for which only the first condition holds. Thus, an open execution may contain some messages that are sent, but not received. This definition is needed later in the paper; an actual execution of the system is never open. Finally, we introduce the following notation for brevity:

[3] Instead of an actual terminal, nodes may have any other local I/O interface that cannot be observed remotely.

- $\textsc{recv}(i,m) \in e$ iff $m$ is a message with $i = \mathit{dest}(m)$ and $(i,I,O) \in e$ with $m \in I$.
- $\textsc{send}(i,m,j) \in e$ iff $m$ is a message with $j = \mathit{dest}(m)$ and $(i,I,O) \in e$ with $m \in O$.
- $\textsc{in}(i,t) \in e$ if $t$ is a terminal input and $(i,I,O) \in e$ with $t \in I$.
- $\textsc{out}(i,t) \in e$ if $t$ is a terminal output and $(i,I,O) \in e$ with $t \in O$.

### 2.2 Algorithms and correctness

Each node $i$ is assigned an *algorithm* $A_i = (M_i, TI_i, TO_i, \Sigma_i, \sigma_0^i, \alpha_i)$, where $M_i$ is the set of messages $i$ can send or receive, $TI_i$ is a set of terminal inputs $i$ can receive, $TO_i$ is a set of terminal outputs $i$ can produce, $\Sigma_i$ is a set of states, $\sigma_0^i \in \Sigma_i$ is the initial state, and $\alpha_i : \Sigma_i \times P(M_i \cup TI_i) \to \Sigma_i \times P(M_i \cup TO_i)$ maps a set of inputs and the current state to a set of outputs and the new state. Here, $P(X)$ denotes the power set of $X$. For convenience, we define $\alpha(\sigma, \emptyset) := (\sigma, \emptyset)$ for all $\sigma \in \Sigma_i$.

We make the following four assumptions about any algorithm $A_i$: a) it only sends messages that can be properly authenticated, b) it never sends the same message twice, c) it discards incoming duplicates and any messages that cannot be authenticated, and d) it never delegates the ability to send messages $m$ with $\mathit{src}(m) = i$, e.g., by revealing or leaking $i$'s key material. Note that assumption b) does not affect generality, since $A_i$ can simply include a nonce with each message it sends. We also assume that it is possible to decide whether $A_i$, starting from some state $\sigma_x$, could receive some set of messages $X$ in any order (plus an arbitrary number of terminal inputs) without sending any messages. This trivially holds if $|\Sigma_i| < \infty$.

We say that a node $i$ is *correct* in execution $e|_i = (i, I_1, O_1), (i, I_2, O_2), \ldots$ with respect to an algorithm $A_i$ iff there is a sequence of states $\sigma_0, \sigma_1, \ldots$ in $\Sigma_i$ such that $\sigma_0 = \sigma_0^i$ and, for all $k \geq 1$, $\alpha_i(\sigma_{k-1}, I_k) = (\sigma_k, O_k)$. Note that correctness of a node $i$ implies that the node is *live*: if $i$ is in a state $\sigma_{k-1}$ and receives an input $I$, then $i$ must produce an output $O_k$ such that $\alpha_i(\sigma_{k-1}, I_k) = (\sigma_k, O_k)$. If $i$ is not correct in $e|_i$ with respect to $A_i$, we say that $i$ is *faulty* in $e|_i$ with respect to $A_i$.

A *distributed algorithm* is a tuple $(A_1, \ldots, A_{|N|})$, one algorithm per node, such that $M_i = M_j$ for all $i, j$. When we say that an execution $e$ is an execution of a distributed algorithm $A$, this implies that each node $i$ is considered correct or faulty in $e$ with respect to the algorithm $A_i$ it has been assigned. We write $\mathit{corr}(A, e)$ to denote the set of nodes that are correct in $e$ with respect to $A$.

### 2.3 Extensions

$(\bar{A}, A, \mu_m, \mu_s, XO)$ is called a *reduction* of an algorithm $\bar{A} = (\bar{M}, \bar{TI}, \bar{TO}, \bar{\Sigma}, \bar{\sigma}_0, \bar{\alpha})$ to an algorithm $A = (M, TI, TO, \Sigma, \sigma_0, \alpha)$ iff $\mu_m$ is a total map $\bar{M} \mapsto P(M)$, $\mu_s$ is a total map $\bar{\Sigma} \mapsto \Sigma$, and the following conditions hold:

X1 $\bar{TI} = TI$, that is, $A$ accepts the same terminal inputs as $\bar{A}$;
X2 $\bar{TO} = TO \cup XO$ and $TO \cap XO = \emptyset$, that is, $A$ produces the same terminal outputs as $\bar{A}$, except $XO$;
X3 $\mu_s(\bar{\sigma}_0) = \sigma_0$, that is, the initial state of $\bar{A}$ maps to the initial state of $A$;
X4 $\forall m \in M\ \exists \bar{m} \in \bar{M} : \mu_m(\bar{m}) = m$, that is, every message of $A$ has at least one counterpart in $\bar{A}$;
X5 $\forall \sigma \in \Sigma\ \exists \bar{\sigma} \in \bar{\Sigma} : \mu_s(\bar{\sigma}) = \sigma$, that is, every state of $A$ has at least one counterpart in $\bar{\Sigma}$;
X6 $\forall \bar{\sigma}_1, \bar{\sigma}_2 \in \bar{\Sigma}, \bar{mi}, \bar{mo} \subseteq \bar{M}, ti \subseteq TI, to \subseteq TO : [\bar{\alpha}(\bar{\sigma}_1, \bar{mi} \cup ti) = (\bar{\sigma}_2, \bar{mo} \cup to)] \Rightarrow [\alpha(\mu_s(\bar{\sigma_1}), \mu_m(\bar{mi}) \cup ti) = (\mu_s(\bar{\sigma_2}), \mu_m(\bar{mo}) \cup (to \setminus XO))]$, that is, there is a homomorphism between $\bar{\alpha}$ and $\alpha$.

If there exists at least one reduction from an algorithm $\bar{A}$ to an algorithm $A$, we say that $\bar{A}$ is an *extension* of $A$. For every reduction $(\bar{A}, A, \mu_m, \mu_s, XO)$ we can construct an *execution mapping* $\mu_e$ that maps executions of $\bar{A}$ to (possibly open) executions of $A$ as follows:

1. Start with $e = \emptyset$.
2. For each new event $(i, \bar{I}, \bar{O})$, perform the following steps:
   (a) Compute $I := (\bar{I} \cap TI_i) \cup \mu_m(\bar{I} \cap \bar{M})$ and $O := (\bar{O} \cap TO_i) \cup \mu_m(\bar{O} \cap \bar{M})$.
   (b) Remove from $I$ any $m \in M$ with $dest(m) \neq i$ or $\textsc{recv}(i, m) \in e$.
   (c) Remove from $O$ any $m \in M$ with $\textsc{send}(i, m, j) \in e$.
   (d) For each node $j \in N$, compute $O_j := \{m \in O \mid src(m) = j\}$.
   (e) If $I \neq \emptyset$ or $O_i \neq \emptyset$, append $(i, I, O_i)$ to $e$.
   (f) For each $j \neq i$ with $O_j \neq \emptyset$, append $(j, \emptyset, O_j)$ to $e$.

A simple example of a reduction is the identity $(A, A, id, id, \emptyset)$. Note that there is a syntactic correspondence between an extension and its original algorithm, not just a semantic one. In other words, the extension not only solves the same problem as the original algorithm (by producing the same terminal outputs as the original), it also solves it in the same way (by sending the same messages in the same order). Recall that our goal is to detect whether or not the nodes in the system are following a given algorithm; we are *not* trying to find a better algorithm. Next, we state a few simple lemmas about extensions.

**Lemma 1.** *Let $\bar{A}$ and $A$ be two algorithms for which a reduction $(\bar{A}, A, \mu_m, \mu_s, XO)$ exists. Then, if $\bar{e}$ is an execution in which a node $i$ is correct with respect to $\bar{A}$, $i$ is correct in $\mu_e(\bar{e})$ with respect to $A$.*

Note that, if a node $i$ is correct in $\bar{e}$ with respect to $\bar{A}$, then it must be correct in $\mu_e(\bar{e})$ with respect to $A$, but the reverse is not true. In other words, it is possible for a node $i$ to be faulty in $\bar{e}$ with respect to $\bar{A}$ but still be correct in $\mu_e(\bar{e})$ with respect to $A$.

**Lemma 2.** *Let $\bar{A}$ and $A$ be two algorithms for which a reduction $(\bar{A}, A, \mu_m, \mu_s, XO)$ exists, let $\bar{e}_1$ be an execution of $\bar{A}$, and let $\bar{e}_2$ be a prefix of $\bar{e}_1$. Then $\mu_e(\bar{e}_2)$ is a prefix of $\mu_e(\bar{e}_1)$.*

**Lemma 3.** *Let $\bar{A}$ and $A$ be two algorithms for which a reduction $(\bar{A}, A, \mu_m, \mu_s, XO)$ exists, and let $e$ be an execution of $A$. Then there exists an execution $\bar{e}$ of $\bar{A}$ such that a) $\mu_e(\bar{e}) = e$ (modulo duplicate messages sent by faulty nodes in $e$), and b) a node $i$ is correct in $\bar{e}$ with respect to $\bar{A}$ iff it is correct in $e$ with respect to $A$.*

### 2.4 Facts and evidence

To detect faults, and to identify faulty nodes, the correct nodes must collect information about the current execution. Clearly, no correct node can expect to know the entire execution at any point, since it cannot observe events on other nodes. However, each node can locally observe its inputs and outputs, and each input or output rules out some possible executions that *cannot* be the current execution. For example, if a node $i$ receives a message $m$, this rules out all executions in which $m$ was never sent. If $i$ manages to rule out all executions in which some set $S$ of nodes is correct, it has established that at least one node $s \in S$ must be faulty. Thus, we can use sets of plausible executions to represent a node's knowledge about the current execution.

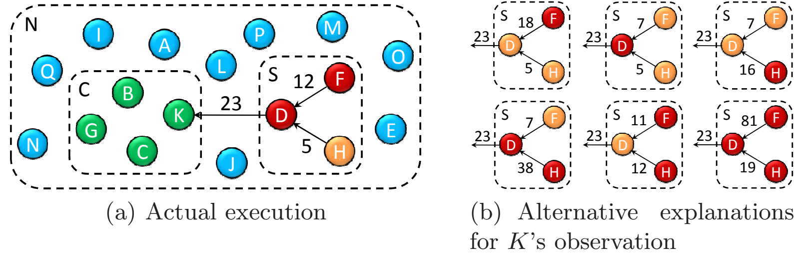
Formally, we define a *fact* $\zeta$ to be a set of executions, and we say that a node $i$ *knows* a fact $\zeta$ at the end of an execution prefix $e$ iff $\zeta$ contains all infinite executions $e'$ where $e|_i$ is a prefix of $e'|_i$ (in other words, $e'$ is consistent with all the inputs and outputs $i$ has seen in $e$). If a node knows two facts $\zeta_1$ and $\zeta_2$, it can combine them into a new fact $\zeta_3 := \zeta_1 \cap \zeta_2$. If the system is running an extension $\bar{A}$ of an algorithm $A$, we can map any fact $\bar{\zeta}$ about the current execution $\bar{e}$ of $\bar{A}$ to a fact $\zeta := \{\mu_e(x) \,|\, x \in \bar{\zeta}\}$ about $\mu_e(\bar{e})$.

Different nodes may know different facts. Hence, the nodes may only be able to detect a fault if they exchange information. However, faulty nodes can lie, so a correct node can safely accept a fact from another node only if it receives *evidence* of that fact. Formally, we say that a message $m$ is evidence of a fact $\zeta$ iff for any execution $\bar{e}$ of $\bar{A}$ in which any node receives $m$, $\mu(\bar{e}) \in \zeta$. Intuitively, evidence consists of signed messages. For more details, please see Section 4.

### 2.5 Fault instances and fault classes

Not all faults can be detected, and some extensions can detect more faults than others. To quantify this, we introduce an abstraction for an individual 'fault'. A *fault instance* $\psi$ is a four-tuple $(A, C, S, e)$, where $A$ is a distributed algorithm, $C$ and $S$ are sets of nodes, and $e$ is an infinite execution, such that a) $C$ and $S$ do not overlap, b) every $c \in C$ is correct in $e$ with respect to $A$, and c) at least one node $i \in S$ is faulty in $e$ with respect to $A$. A *fault class* $F$ is a set of fault instances, and the nodes in $S$ are called *suspects*.

Intuitively, the goal is for the correct nodes in $C$ to identify at least one faulty suspect from $S$. Of course, an ideal solution would simply identify *all* the nodes that are faulty in $e$ with respect to $A$; however, this is not always possible. Consider the scenario in Figure 1. In this scenario, the nodes in $C$ know that at least one of the nodes in $S$ must be faulty, but they do not know which ones, or how many. Thus, the size of the set $S$ effectively represents the precision with




(a) Actual execution

(b) Alternative explanations for $K$'s observation

**Fig. 1.** Example scenario. Nodes $F$ and $H$ are supposed to each send a number between 1 and 10 to $D$, who is supposed to add the numbers and send the result to $K$. If $K$ receives 23, it knows that at least one of the nodes in $S = \{D, F, H\}$ must be faulty, but it does not know which ones, or how many.

which the fault can be localized. The best case is $|S| = 1$; this indicates that the fault can be traced to exactly one node. The worst case is $S = N \setminus C$; this indicates that the nodes in $C$ know that a fault exists somewhere, but they are unable to localize it.

### 2.6 Environments

Our formulation of the fault detection problem does not require a bound on the number of faulty nodes. However, if such a bound is known, it is possible to find solutions with a lower message complexity. To formalize this, we use the notion of an *environment*, which is a restriction on the fault patterns that may occur in a system. In this paper, we specifically consider environments $E_f$, in which the total number of faulty nodes is limited to $f$. If a system in environment $E_f$ is assigned a distributed algorithm $A$, the only executions that can occur are those in which at most $f$ nodes are faulty with respect to $A$.

## 3 The fault detection problem

Let $\nu := \{\textsc{faulty}(X) \,|\, X \subseteq N\}$ be a set of *fault notifications*. Then the *fault detection problem* for a fault class $F$ is to find a transformation $\tau_F$ that maps any distributed algorithm $A$ to an extension $\bar{A} := \tau_F(A)$ such that $\bar{TO} = TO \cup \nu$ and the following conditions hold:

C1 **Nontriviality:** If $\bar{e}$ is an infinite execution of $\bar{A}$ and $i \in N$ is correct in $\bar{e}$ with respect to $\bar{A}$, then $i$ outputs infinitely many fault notifications in $\bar{e}$.

C2 **Completeness:** If $(A, C, S, e)$ is a fault instance in $F$, $\bar{e}$ is an infinite execution such that $\mu_e(\bar{e}) = e$, and each node $c \in C$ is correct in $\bar{e}$ with respect to $\bar{A}$, then there exists a correct node $c' \in N$ and a node $j \in S$ such that eventually all fault notifications output by $c'$ contain $j$.

C3 **Accuracy:** If $\bar{e}$ is an infinite execution of $\bar{A}$ and $c_1, c_2 \in N$ are any two nodes that are correct in $\bar{e}$ with respect to $\bar{A}$, then $c_1$ outputs infinitely many fault notifications that do not include $c_2$.

We also consider the *fault detection problem with agreement*, which additionally requires:

C4 **Agreement:** If $c_1 \in N$ and $c_2 \in N$ are correct in an execution $\bar{e}$ with respect to $\bar{A}$ and there exists a node $i \in N$ such that eventually all fault notifications output by $c_1$ in $\bar{e}$ include some node $i \in N$, then eventually all fault notifications output by $c_2$ in $\bar{e}$ include $i$ as well.

Note that condition C2 does not require us to detect nodes that are faulty in $\bar{e}$ with respect to $\bar{A}$, but correct in $\mu_e(\bar{e})$ with respect to $A$. Thus, we avoid the infinite recursion that would result from trying to detect faults in the detector itself. Note also that condition C3 is weaker than the definition of eventual strong accuracy in [6], which requires that correct nodes eventually output only faulty nodes. This change is necessary to make the problem solvable in an asynchronous environment.

## 4 Which faults can be detected?

In the rest of this paper, we assume that the only facts for which evidence can exist are a) message transmissions, and b) message receptions. Specifically, a properly authenticated message $\bar{m}$ with $\mu_m(\bar{m}) = m$ and $src(m) = i$ in an execution $\bar{e}$ is evidence of a fact $\{e \,|\, \text{SEND}(i, m, dest(m)) \in e\}$ about $\mu_e(\bar{e})$, and a properly authenticated message $\bar{m}'$ with $src(\bar{m}') = i$, $m \in \bar{m}'$, and $dest(m) = i$ in an execution $\bar{e}$ is evidence of a fact $\{e \,|\, \text{RECV}(i, m) \in e\}$ about $\mu_e(\bar{e})$. Note that in some systems it may be possible to construct evidence of additional facts (e.g., when the system has more synchrony or access to more sophisticated cryptographic primitives). In such systems, the following results may not apply.

### 4.1 Definitions

We define two *fact maps* $\phi^+$ and $\phi^-$ as follows. Let $e$ be an infinite execution or an execution prefix, and let $C$ be a set of nodes. Then $\phi^+(C, e)$ is the intersection[4] of all facts $\zeta$ for which at least one node in $C$ can construct evidence in $e$ (note that there is usually no single node that can construct evidence of *all* facts), and $\phi^-(C, e)$ is the intersection of all facts $\zeta$ such that, if the complement $\bar{\zeta}$ were a fact in $e$ (i.e., $e \in \zeta$), then at least one node in $C$ could construct evidence of $\bar{\zeta}$ in $e$, but $\bar{\zeta} \notin \phi^+(C, e)$. For brevity, we write $\phi^\pm(C, e)$ to represent both kinds of facts, that is, $\phi^\pm(C, e) := \phi^+(C, e) \cap \phi^-(C, e)$.

Intuitively, $\phi^\pm$ represents the sum of all knowledge the nodes in $C$ can have in $e$ if they exchange all of their evidence with each other. Since we have restricted the admissible evidence to messages earlier, $\phi^+(C, e)$ effectively represents knowledge about all the messages sent or received in $e$ by the nodes in $C$,

[4] Recall that facts are combined by forming the intersection. Since facts are sets of plausible executions, an execution that is plausible given two facts $\zeta_1$ and $\zeta_2$ must be a member of $\zeta_1 \cap \zeta_2$.

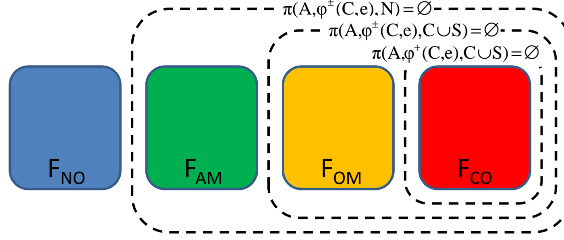



**Fig. 2.** Classification of all fault instances. The fault detection problem cannot be solved for fault instances in $F_{NO}$ (Theorem 2) or $F_{AM}$ (Theorem 3), but solutions exist for $F_{OM}$ and $F_{CO}$ (Theorem 4).

while $\phi^-(C, e)$ effectively represents knowledge about all the messages *not* sent or received in $e$ by the nodes in $C$.

We also define the *plausibility map* $\pi$ as follows. Let $A$ be a distributed algorithm, $Z$ a set of facts, and $C$ a set of nodes. Then $\pi(A, Z, C)$ represents all infinite executions $e \in Z$ in which each node $c \in C$ is correct in $e$ with respect to $A$. Intuitively, $\pi(A, Z, C)$ is the set of executions of $A$ that are plausible given the facts in $Z$, and given that (at least) the nodes in $C$ are correct.

A few simple properties of $\phi$ and $\pi$ are: 1) $C_1 \subseteq C_2 \Rightarrow \phi(C_2, e) \subseteq \phi(C_1, e)$, that is, adding evidence from more nodes cannot reduce the overall knowledge; 2) $p_1 \upharpoonright p_2 \Rightarrow \phi(C, p_2) \subseteq \phi(C, p_1)$, that is, knowledge can only increase during an execution; 3) $C_1 \subseteq C_2 \Rightarrow \pi(A, Z, C_2) \subseteq \pi(A, Z, C_1)$, that is, assuming that more nodes are correct can only reduce the number of plausible executions; and 4) $Z_1 \subseteq Z_2 \Rightarrow \pi(A, Z_1, C) \subseteq \pi(A, Z_2, C)$, that is, learning more facts can only reduce the number of plausible executions.

### 4.2 Fault classes

We define the following fault classes (see also Figure 2):

$$
\begin{aligned}
F_{NO} &:= \{(A, C, S, e) \mid \pi(A, \phi^{\pm}(C, e), N) \neq \emptyset\} \\
F_{AM} &:= \{(A, C, S, e) \mid \pi(A, \phi^{\pm}(C, e), N) = \emptyset \wedge \pi(A, \phi^{\pm}(C, e), C \cup S) \neq \emptyset\} \\
F_{OM} &:= \{(A, C, S, e) \mid \pi(A, \phi^{\pm}(C, e), C \cup S) = \emptyset \wedge \pi(A, \phi^{+}(C, e), C \cup S) \neq \emptyset\} \\
F_{CO} &:= \{(A, C, S, e) \mid \pi(A, \phi^{+}(C, e), C \cup S) = \emptyset\}
\end{aligned}
$$

$F_{NO}$ is the class of *non-observable faults*. For executions in this class, the nodes in $C$ cannot even be sure that the system contains any faulty nodes, since there exists a correct execution of the entire system that is consistent with everything they see. We will show later in this section that the fault detection problem cannot be solved for faults in this class.

$F_{AM}$ is the class of *ambiguous fault instances*. When a fault instance is in this class, the nodes in $C$ know that a faulty node exists, but they cannot be sure that it is one of the nodes in $S$. We will show later that the fault detection problem cannot be solved for fault instances in this class. Note that the problem

here is not that the faults cannot be observed from $C$, but that the set $S$ is too small. If $S$ is sufficiently extended (e.g., to $N \setminus C$), these fault instances become solvable.

$F_{OM}$ is the class of *omission faults*. For executions in this class, the nodes in $C$ could infer that one of the nodes in $S$ is faulty if they knew all the facts, but the positive facts alone are not sufficient; that is, they would also have to know that some message was *not* sent or *not* received. Intuitively, this occurs when the nodes in $S$ refuse to send some message they are required to send.

$F_{CO}$ is the class of *commission faults*. For executions in this class, the nodes in $C$ can infer that one of the nodes in $S$ is faulty using only positive facts. Intuitively, this occurs when the nodes in $S$ send some combination of messages they would never send in any correct execution.

**Theorem 1.** *$(F_{NO}, F_{AM}, F_{OM}, F_{CO})$ is a partition of the set of all fault instances.*

*Proof.* First, we show that no fault instance can belong to more than one class. Suppose $\psi := (A, C, S, e) \in F_{NO}$; that is, there is a plausible correct execution $e'$ of the entire system. Then $\psi$ can obviously not be in $F_{AM}$, since $\pi(A, \phi^{\pm}(C, e), N)$ cannot be both empty and non-empty. Since all nodes are correct in $e'$, the nodes in $C \cup S$ in particular are also correct, so $\psi \notin F_{OM}$ (Section 4.1, Property 3), and they are still correct if negative facts are ignored, so $\psi \notin F_{CO}$. Now suppose $\psi \in F_{AM}$. Obviously, $\psi$ cannot be in $F_{OM}$, since $\pi(A, \phi^{\pm}(C, e), C \cup S)$ cannot be both empty and non-empty. But $\psi$ cannot be in $F_{CO}$ either, since using fewer facts can only increase the number of plausible executions (Section 4.1, Property 1). Finally, observe that $\psi$ cannot be in both $F_{OM}$ and $F_{CO}$, since $\pi(A, \phi^{+}(C, e), C \cup S)$ cannot be both empty and non-empty.

It remains to be shown that any fault instance belongs to at least one of the four classes. Suppose there is a fault instance $\psi \notin (F_{NO} \cup F_{AM} \cup F_{OM} \cup F_{CO})$. Since $\psi$ is not in $F_{NO}$, we know that $\pi(A, \phi^{\pm}(C, e), N) = \emptyset$. But if this is true and $\psi$ is not in $F_{AM}$, it follows that $\pi(A, \phi^{\pm}(C, e), C \cup S) = \emptyset$. Given this and that $\psi$ is not in $F_{OM}$, we can conclude that $\pi(A, \phi^{+}(C, e), C \cup S) = \emptyset$. But then $\psi$ would be in $F_{CO}$, which is a contradiction.

**Theorem 2.** *The fault detection problem cannot be solved for any fault class $F$ with $F \cap F_{NO} \neq \emptyset$.*

*Proof sketch.* The proof works by showing that, for any fault instance $\psi := (A, C, S, e) \in F_{NO}$, we can construct two executions $\bar{e}_{good}$ and $\bar{e}_{bad}$ of $\bar{A} := \tau(A)$ such that a) all the nodes are correct in $\bar{e}_{good}$, b) the fault occurs in $\bar{e}_{bad}$, and c) the two executions are indistinguishable from the perspective of the nodes in $C$ (that is, $\bar{e}_{good}|_C = \bar{e}_{bad}|_C$). Hence, the nodes in $C$ would have to both expose some node in $S$ (to achieve completeness in $\bar{e}_{bad}$) and *not* expose any node in $S$ (to achieve accuracy in $\bar{e}_{good}$) based on the same information, which is impossible. For the full proof, see [9]. □

**Theorem 3.** *The fault detection problem cannot be solved for any fault class $F$ with $F \cap F_{AM} \neq \emptyset$.*

*Proof sketch.* The proof is largely analogous to that of Theorem 2, except that we now construct two executions $\bar{e}_{\in S}$ and $\bar{e}_{\notin S}$ of $\bar{A} := \tau(A)$ such that a) in $\bar{e}_{\in S}$ the faulty node is a member of $S$, b) in $\bar{e}_{\notin S}$ all the nodes in $S$ are correct, and c) the two executions are indistinguishable from $C$. For the full proof, see [9]. □

**Corollary 1.** *If the fault detection problem can be solved for a fault class $F$, then $F \subseteq F_{OM} \cup F_{CO}$.*

**Theorem 4.** *There is a solution to the fault detection problem with agreement for the fault class $F_{OM} \cup F_{CO}$.*

For a transformation that solves the fault detection problem for this class, please refer to the proof of Theorem 8 (Section 5.2) that appears in [9].

## 5 Message complexity

In this section, we investigate how expensive it is to solve the fault detection problem, that is, how much additional work is required to detect faults. The metric we use is the number of messages that must be sent by correct nodes. (Obviously, the faulty nodes can send arbitrarily many messages). Since the answer clearly depends on the original algorithm and on the actions of the faulty nodes in a given execution, we focus on the following two questions: First, what is the maximum number of messages that may be *necessary* for some algorithm, and second, what is the minimum number of messages that is *sufficient* for any algorithm?

### 5.1 Definitions

If $\tau$ is a solution of the fault detection problem, we say that the *message complexity* $\gamma(\tau)$ of $\tau$ is the largest number such that for all $k$, there exists an algorithm $A$, an execution $e$ of $A$, and an execution $\bar{e}$ of $\tau(A)$ such that

$$(\mu_e(\bar{e}) = e) \wedge (|e| \geq k) \wedge \left[ \frac{|\{\bar{m} \mid \textsc{send}(i, \bar{m}, j) \in \bar{e} \wedge i \in \mathit{corr}(\tau(A), \bar{e})\}|}{|e|} \geq \gamma(\tau) \right]$$

In other words, the message complexity is the maximum number of messages that must be sent by correct nodes in any $\bar{e}$ *per* message sent in the corresponding $e := \mu_e(\bar{e})$. The message complexity of the fault detection problem as a whole is the minimum message complexity over all solutions.

### 5.2 Lower and upper bounds

In this section, we present a collection of tight lower bounds for solving various instances of the fault detection problem. Omitted proofs can be found in the technical report [9].

First we show that message complexity of the fault detection problem in the environment $E_f$ for both commission and omission faults is optimally linear in $f$.

**Theorem 5.** *Any solution* $\tau$ *of the fault detection problem for* $F_{CO}$ *in the environment* $E_f$ *has message complexity* $\gamma(\tau) \geq f + 2$*, provided that* $f + 2 < |N|$*.*

**Theorem 6.** *The message complexity of the fault detection problem with agreement for* $F_{CO}$ *in the environment* $E_f$ *is at most* $f + 2$*, provided that* $f + 2 < |N|$*.*

**Corollary 2.** *The message complexity of the fault detection problem (with or without agreement) for* $F_{CO}$ *in environment* $E_f$ *is* $f+2$*, provided that* $f+2 < |N|$*.*

**Theorem 7.** *Any solution* $\tau$ *of the fault detection problem for* $F_{OM}$ *in the environment* $E_f$ *has message complexity* $\gamma(\tau) \geq 3f + 4$*, provided that* $f + 2 < |N|$*.*

**Theorem 8.** *The message complexity of the fault detection problem for* $F_{OM}$ *in the environment* $E_f$ *is at most* $3f + 4$*, provided that* $f + 2 < |N|$*.*

Interestingly, if we additionally require agreement, then the optimal message complexity of the fault detection problem with respect to omission faults is quadratic in $|N|$, under the condition that at least half of the nodes may fail. Intuitively, if a majority of $N$ is known to be correct, it should be possible to delegate fault detection to a set $\omega$ with $|\omega| = 2f + 1$, and to have the remaining nodes follow the majority of $\omega$. This would reduce the message complexity to approximately $|N| \cdot (2f + 1)$.

**Theorem 9.** *Any solution* $\tau$ *of the fault detection problem with agreement for* $F_{OM}$ *in the environment* $E_f$ *has message complexity* $\gamma(\tau) \geq (|N|-1)^2$*, provided that* $\frac{|N|-1}{2} < f < |N| - 2$*.*

*Proof sketch.* In contrast to commission faults, there is no self-contained proof of an omission fault; when a node is suspected of having omitted a message $m$, the suspicion can always turn out to be groundless when $m$ eventually arrives. We show that, under worst-case conditions, such a 'false positive' can occur after every single message. Moreover, since agreement is required, a correct node must not suspect (or stop suspecting) another node unless every other correct node eventually does so as well. Therefore, after each message, the correct nodes may have to ensure that their own evidence is known to all the other correct nodes, which in the absence of a correct majority requires reliable broadcast and thus at least $(|N| - 1)^2$ messages. For the full proof, see [9]. □

**Theorem 10.** *The message complexity of the fault detection problem with agreement for* $F_{OM}$ *in the environment* $E_f$ *is at most* $(|N| - 1)^2$*, provided that* $f + 2 < |N|$*.*

### 5.3 Summary

Table 1 summarizes the results in this section. Our two main results are that a) detecting omission faults has a substantially higher message complexity than detecting commission faults, and that b) the message complexity is generally linear in the failure bound $f$, except when the fault class includes omission faults *and* agreement is required, in which case the message complexity is quadratic in the system size $|N|$.

| Fault class | Fault detection problem | Fault detection problem with agreement |
|---|---|---|
| $F_{CO}$ | $f+2$ (Corollary 2) | $f+2$ (Corollary 2) |
| $F_{OM}$ | $3f+4$ (Theorems 7 and 8) | $(\|N\|-1)^2$ (Theorems 9 and 10) |

**Table 1.** Message complexity in environments with up to $f$ faulty nodes.

## 6 Related work

There is an impressive amount of work on fault detection in the context of *failure detectors* (starting from the original paper by Chandra and Toueg [6]). However, literature on failure detectors conventionally assumes crash-fault models, and usually studies theoretical bounds on the information about failures that is necessary to solve various distributed computing problems [5], without focusing on the costs of implementing failure detectors.

Faults beyond simple crashes have been extensively studied in the context of arbitrary (Byzantine) fault tolerance (starting from the original paper by Lamport et al. [15]). Byzantine fault-tolerant systems aim to keep faults from becoming "visible" to the system users. One example is Castro and Liskov's Practical Byzantine fault-tolerance (PBFT) [4] that extends Lamport's state-machine replication protocol [14] to the Byzantine failure model. However, BFT systems do not detect and expose faulty nodes.

In the context of *synchronous* Byzantine agreement algorithms, Bar-Noy et al [2] use the terms "fault detections" and "fault masking" in a more restrictive manner than this paper does. In [2], a processor in an agreement protocol is said to be "detected" if all correct processors agree that the processor is faulty. All subsequent actions of this processor are then ignored and thus "masked".

Also with respect to Byzantine agreement algorithms, Bracha [3] describes a protocol in which all messages are broadcast, and in which all nodes track the state of every other node in order to identify messages that could not have been sent by a correct node.

Intrusion detection systems (IDS) can detect a limited class of protocol violations, for example by looking for anomalies [7] or by checking the behavior of the system against a formal specification [13].

A technique that statistically monitors quorum systems and raises an alarm if the failure assumptions are about to be violated was introduced in [1]. However, this technique cannot identify which nodes are faulty.

To the best of our knowledge, Kihlstrom et al. [12] were the first to explicitly focus on Byzantine fault detection. The paper also gives informal definitions of the commission and omission faults. However, the definitions in [12] are specific to consensus and broadcast protocols.

Our notions of facts and evidence in a distributed system are inspired by the epistemic formalism of Halpern and Moses [11].

The results in this paper have important consequences for research on *accountability* in distributed computing. Systems like PeerReview [10] provide accountability by ensuring that faults can eventually be detected and irrefutably linked to a faulty node. Since fault detection is an integral part of accountability, this paper establishes an upper bound on the set of faults for which accountability can be achieved, as well as a lower bound on the worst-case message complexity. Note that practical accountability systems have other functions, such as providing more detailed fault notifications, which we do not model here.

## 7 Conclusion and future work

In reasoning about computing systems, it is very important to find the right language. Somewhat dangerously, intuitive claims sometimes become "folklore" before they are actually stated precisely and proved. For example, exact bounds on the information about crash failures needed for solving agreement, though informally anticipated earlier [8,14], were captured precisely only with the introduction of failure detectors [6], and especially the notion of the weakest failure detector [5].

Similarly, this paper has developed a language for reasoning about fault detection with general fault models (beyond simple crash faults). We have proposed a framework in which generic faults can be precisely defined and classified. Unlike crash faults, generic faults cannot be defined without reference to an algorithm, which is why we have introduced the expected system behavior into the definition. To determine the inherent costs of generic fault detection, we have proposed a weak definition of the fault detection problem, and we have derived exact bounds on the cost of solving it in asynchronous message-passing systems where nodes are able to digitally sign their messages.

The framework we have presented can also be used to study fault detection in other system models. If the model is weakened or strengthened (e.g., by varying the assumptions about the network, the degree of synchrony, or the available cryptographic primitives), the kinds of evidence available to correct nodes can change, as can the set of executions that are plausible given some specific evidence. This change, in turn, affects the ability of correct nodes to detect and isolate faulty nodes. For instance, if bounds on communication and processing times are known, it is possible to establish in finite time that an omission fault has occurred, and the culprits can safely be suspected forever. The model could also be changed by introducing bounds on the message size and/or the set of states $\Sigma$. These changes would likely increase the message complexity and reduce the size of the fault classes for which detection is possible.

Our framework can be used to study different variants of the fault detection problem. The (weak) formulation of the problem chosen in this paper was primarily instrumental for establishing impossibilities and complexity lower bounds that capture inherent costs of detection in the asynchronous systems. In other scenarios, however, different formulations may make more sense. For example, accuracy could be strengthened such that eventually no correct node is suspected

by any correct node; this would require stronger synchrony assumptions [6, 8]. On the other hand, completeness could be relaxed in such a way that faults must only be detected with high probability. Preliminary evidence suggests that such a definition would substantially reduce the message complexity [10].

In conclusion, we believe that this work is a step toward a better understanding of the costs and limitations of fault detection in distributed systems. We also believe that this work could be used as a basis for extending the spectrum of fault classes with new intermediate classes, ranging between the "benign" crash faults (which have proven to be too restrictive for modern software) and the generic but rather pessimistic Byzantine faults.

## References


1. Alvisi, L., Malkhi, D., Pierce, E.T., Reiter, M.K.: Fault detection for Byzantine quorum systems. IEEE Trans. Parallel Distrib. Syst. 12(9), 996–1007 (2001)
2. Bar-Noy, A., Dolev, D., Dwork, C., Strong, H.R.: Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. In: PODC. pp. 42–51 (1987)
3. Bracha, G.: Asynchronous Byzantine agreement protocols. Information and Computation 75(2), 130–143 (Nov 1987)
4. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (Nov 2002)
5. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (Jul 1996)
6. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (Mar 1996)
7. Denning, D.E.: An intrusion-detection model. IEEE Transactions on Software Engineering 13(2), 222–232 (1987)
8. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. J. ACM 34(1), 77–97 (Jan 1987)
9. Haeberlen, A., Kuznetsov, P.: The fault detection problem (Oct 2009), Technical Report MPI-SWS-2009-005, Max Planck Institute for Software Systems
10. Haeberlen, A., Kuznetsov, P., Druschel, P.: PeerReview: Practical accountability for distributed systems. In: SOSP. pp. 175–188 (Oct 2007)
11. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM 37(3), 549–587 (Jul 1990)
12. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine fault detectors for solving consensus. The Computer Journal 46(1), 16–35 (Jan 2003)
13. Ko, C., Fink, G., Levitt, K.: Automated detection of vulnerabilities in privileged programs using execution monitoring. In: Proceedings of the 10th Annual Computer Security Application Conference (Dec 1994)
14. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (May 1998)
15. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Progr. Lang. Syst. 4(3), 382–401 (Jul 1982)
16. Li, J., Krohn, M., Mazières, D., Sasha, D.: Secure untrusted data repository (SUNDR). In: OSDI (Dec 2004)
17. Vandiver, B., Balakrishnan, H., Liskov, B., Madden, S.: Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In: SOSP (Oct 2007)