On the Equivalence of Partial Synchrony and Asynchrony with Unreliable Failure Detectors

Felix Freiling¹, Martin Hutle²
Lucia Draque Penso¹, and Josef Widder²

¹ Computer Science Department, University of Mannheim,
  D-68131 Mannheim, Germany
² Embedded Computing Systems Group, TU Wien
  Treithstraße 3 A-1040 Vienna, Austria

July 4, 2006

Abstract

In this paper we explore whether the information of faults provided by unreliable failure detectors is equivalent to the semantics partial synchrony assumptions provide when considering the solvability of distributed computing problems.

Up to now the relation of fault information and timing has been worked out for the perpetual case: It has been shown in [2] that an asynchronous system augmented with the perfect failure detector $\mathcal{P}$ is weaker than a synchronous system, that is, some time-free problems can be solved in a synchronous system but not in an asynchronous system with $\mathcal{P}$.

Our results show the contrary in the “eventual” case, namely that the asynchronous system augmented with an eventually perfect failure detector $\diamond\mathcal{P}$ and partial synchrony are equivalent in terms of solvability.
1 Introduction

We compare asynchronous systems with an eventually perfect failure detector (FD) $\diamond \mathcal{P}$ [1] to partial synchronous, eventual synchronous and eventual lockstep systems [4], in terms of solvability, and derive equivalence results that contrast with their perpetual counterparts that have been shown to differ in [2].

Since computer networks have emerged, deciding for a specific fault-tolerant system model remains a key issue. Over the last decades, a variety of message-passing models that enable feasible solutions for fundamental non-trivial distributed computing problems such as consensus, reliable broadcast and non-blocking atomic commit have been proposed. Though many of them share some features, it is remarkable that many of them happen to fall into one of the following two categories: either processes are equipped with information about time (timing-based models) or failures are detected more or less reliably (FD-based models) [2].

Timing-based models, are composed by a wide range of choices that vary from the synchronous model (the strongest one) to the asynchronous model (the weakest one), depending on knowledge about approximate time. While in the earlier case, bounds on message delays and relative process speeds must be known, in the latter, such bounds simply do not exist. A collection of intermediate models guaranteeing only partial synchrony, where timing information may be limited is presented in [4, 3]. The second approach, FD-based models, also comprises a variety of possibilities, which depend on the axiomatic properties of the FD that are augmented to the asynchronous system [5] and help detecting failures unreliably. For instance, the perfect FD $\mathcal{P}$, which suspects any process to have crashed iff it has crashed, is the strongest FD according to the hierarchy proposed in [1]. An example of a weaker FD is the eventually perfect FD $\diamond \mathcal{P}$, that eventually suspects any process to have crashed iff it has crashed. More detailed definitions for a whole set of miscellaneous FDs can be found in [1].

Despite appearing to have completely distinct natures, there is a basic relation between the two different approaches: The failure detector properties are often regarded as abstractions of time-outs, in the sense that if some process has not heard from another process for some time it is suspected of being crashed. Based on these quite natural underlying idea, the perfect failure detector $\mathcal{P}$ can be implemented using time-outs from the synchronous model, whereas an eventually perfect failure detector $\diamond \mathcal{P}$ can be implemented using (adaptive) time-outs from partial synchronous models [1]. Hence, the challenge in comparing both approaches, in order to choose the best distributed system model option, resides in identifying: (1) if there are characteristics of failure detectors that cannot be translated from timing assumptions and (2) if anything is lost when translating timing assumptions into axiomatic prop-
erties of failure detectors. Both are basic questions on which the practical importance of failure detectors relies.

**Related Work.** Charron-Bost et al. [2] compare the (timing-based) synchronous model, noted $S_S$, with the failure-detector-based model $S_P$, i.e., the asynchronous model augmented with the perfect failure detector. A time-free problem crucial for atomic commit—the strongly dependent decision problem—is shown to be solvable in $S_S$ but not in $S_P$. Thus, the synchronous model indeed contains more computational expressiveness that the asynchronous system augmented with the perfect failure detector $P$.

Later it would be discovered that, in terms of solvability of time-free problems for crash prone systems that the synchronous system is in fact equivalent to an asynchronous model extended with a failure detector sequencer [7], which is strictly stronger than $S_P$. Another example of a time-free problem solvable in $S_S$ but not in $S_P$, namely strict termination detection, can be found in [8].

**Contribution.** In this paper it is shown that in contrast to the results of [2], $\diamond P$ and partial synchrony are equivalent regarding solvable problems. To this end we introduce simulations that allow that every crash tolerant algorithm designed for the partially synchronous system can correctly be executed in the asynchronous system with failure detector. In order to make these simulations, we introduce a general (structural) model from which we derive the models to be compared.

**Roadmap.** The remainder of this paper is organized as follows. In Section 2, definitions for the different system models are introduced. Section 3 shows our main results, that is the simulation of models. We conclude in Section 4 with some final remarks.

**2 System Models**

We assume that all system models in this paper consist of a distributed system of $n$ processes $\Pi = \{p_1 \ldots p_n\}$, which proceed according to a distributed algorithm in steps, until they fail by crashing. A distributed algorithm for a particular system model is a set of $n$ automata, one for each process, composed by local memory and states which undergo a state transition within a computing step whose operations depend on the system model. A process that executes correctly its automaton is correct unless it crashes, i.e., halts prematurely.

Processes communicate in a point to point way by exchanging messages from an alphabet $M$ through a reliable channel, by means of send and receive operations performed during steps. The system models differ in (1) which
operations (send, receive, etc...) may be performed during an step and (2) the types of assumptions that link together operations.

The various models are characterized by the set of executions that are admissible for that model. In particular, in all system models here presented, an execution always fulfills the following properties\(^1\):

**Reliability.** Every message sent to a correct process is eventually delivered.

**Integrity.** A message is delivered only if it was actually sent.

**No Duplication.** No message is received more than once.

Note that, though the **Reliability**, **Integrity**, and **No Duplication** properties must be satisfied for an execution of any of our models, they relate to distinct send and receive primitives, according to the specific system models to be defined in the following sections.

We consider the existence of a discrete global timebase with instants \( t \in \mathbb{N} \), which is inaccessible to processes. At every instant of this time, processes may execute at most one step. The bound \( \Delta \) is said to hold provided that, if a message \( m \) is sent to \( p \) by \( q \) at time \( t_1 \) and \( p \) performs a receive operation at time \( t_2 \geq t_1 + \Delta \), \( m \) is delivered to \( p \) by time \( t_2 \). Moreover, the bound \( \Phi \) is said to hold if in every window of \( \Phi \) consecutive time instants every correct process takes at least one step.

Note that we do not assume that there is some specific mapping of our time base to Newtonian real-time. Specifically we do not restrict the real-time duration of one of our steps. This allows us to establish our synchrony requirements even in asynchronous systems: We will show that the steps of the upper layer algorithm eventually satisfy \( \Delta = 0 \) and \( \Phi = 1 \) while the real-time durations of these steps may vary and just require to be finite. Thus the time base is algorithmically simulated such that consistent cuts separate the time instants. This also includes that the \( i^{th} \) step of our timebase does not need to happen at the same real-time instant at all processes. The simulations just ensure that it appears to the upper layer algorithm as if it would be the case.

### 2.1 Partial Synchrony and Eventual Synchrony

In partial synchrony or eventual synchrony [4], a process \( p \) that runs a valid algorithm executes in every step one of the following operations:

- \( s\text{-send}_p(m, q) \): Sends a message \( m \) to process \( q \).
- \( s\text{-receive}_p() \): Delivers a set \( \emptyset \subseteq S \subseteq M \times \Pi \) of messages sent to \( p \).

\(^1\)Note that we did not choose the buffer representation as in [4] but equivalent separate properties instead to characterize reliable channels.
Hence, the distributed algorithm determines in which order operations are performed. Moreover, we define the following properties:

**Partial Synchronous Communication.** $\Delta$ holds eventually and is unknown.\(^2\)

**Partial Synchronous Processes.** $\Phi$ holds eventually and is unknown.

**Eventual Synchronous Communication** $\Delta$ holds eventually and is known.

**Synchronous Processes.** $\Phi$ holds perpetually and is known.

We denote the system model where all executions fulfill Reliability, Integrity, No Duplication, Partial Synchronous Communication, and Partial Synchronous Processes, as $\text{ParSync}$, whereas we denote the system model where all executions fulfill Reliability, Integrity, No Duplication, Eventual Synchronous Communication and Synchronous Processes, as $\bowtie\text{Synch}$.\(^3\) Note that $\text{ParSync}$ is by definition not stronger than $\bowtie\text{Synch}$, that is, every execution in $\bowtie\text{Synch}$ is also an execution in $\text{ParSync}$.

### 2.2 Eventual Lockstep

In an eventual lockstep system, processes proceed in rounds $r = 0, 1, 2, \ldots$. For a valid algorithm, in every round $r$, a process $p$ executes exactly one step comprising a send operation followed by exactly one step comprising a receive operation, where the operations are defined as:

- $ls\text{-send}_{p}(S, r)$: Sends a set $S \subseteq M \times \Pi$ of messages. For every process $q$, $S$ contains at most one message $m_q$.

- $ls\text{-receive}_{p}(r)$: Delivers a set $\emptyset \subseteq S \subseteq M \times \Pi \times \mathbb{N}$ of messages to $p$, where a tuple $(m, q, r^0)$ denotes a message $m$ sent by $q$ to $p$ in round $r^0$.

Further, we define the property:

**Eventual Lockstep.** There is a round $r'$, such that for all rounds $r \geq r'$, every message $(m, q, r) \in S$ sent via a $ls\text{-send}_{p}(S, r)$ is delivered upon $ls\text{-receive}_{q}(r)$.

We denote the system model where all executions fulfill Reliability, Integrity, No Duplication and Eventual Lockstep, as $\bowtie\text{LS}$.

---

\(^2\)In [4] there is another variant of partial synchrony, where $\Delta$ holds always but is unknown. Since we have reliable channels, this is equivalent to the Partial Synchrony property.

\(^3\)Note that we use a stronger version of eventual synchrony that has synchronous processes instead of eventual synchronous processes in order to strengthen our results.
2.3 Asynchrony with $\Diamond P$

In an asynchronous system model with an eventually perfect failure detector $\Diamond P$, a process $p$ that executes a valid algorithm may execute during every step the following operations in the given order:

- $a$-receive$_p()$: Delivers a message $m$, $\langle m, q \rangle \in M \times \Pi$, sent to $p$ from $q$.
- $a$-queryFD$_p()$: Queries the failure detector of $p$ and returns a set of suspected processes to $p$.
- $a$-send$_p(m, q)$: Sends a message $m$ to process $q$.

Note that not all of these operations (but at least one of them) has to be performed. According to [1], we have the properties of an eventual perfect failure detector $\Diamond P$:

**Strong Completeness.** Eventually every process that crashes is permanently suspected by every correct process.

**Eventual Strong Accuracy.** There is a time after which correct processes are not suspected by any correct process.

We denote the system model where all executions fulfill Reliability, Integrity, No Duplication, Strong Completeness, and Eventual Strong Accuracy, as Async+$\Diamond P$.

3 Solvability Equivalence of System Models

We show the solvability equivalence of system models by reduction. In more detail, we implement the operations of the simulated model in the simulation model. We denote the fact that a model $A$ can be reduced by such an implementation to a model $B$ by $A \preceq B$.

In Section 3.1 we reduce eventual lockstep to asynchrony with $\Diamond P$, in Section 3.2 we reduce eventual synchrony to eventual lockstep, and finally, since partial synchrony is trivially reduced to eventual synchrony by proper inclusion of admissible executions, we reduce asynchrony with $\Diamond P$ to partial synchrony in Section 3.3.

3.1 Reducing Eventual Lockstep to Asynchrony with $\Diamond P$

In this section we show that an arbitrary algorithm $A$ for $\Diamond LS$ can correctly be executed in an Async+$\Diamond P$ system model, by giving implementations of $ls$-send and $ls$-receive in Async+$\Diamond P$. 
Algorithm 1 Reducing $\diamondsuit$LS to Async+\diamondsuitP, code for a process $p$.

1: variables
2: $\text{undelivered}_p \subseteq \mathbb{N} \times M \times \Pi$, initially empty

3: operation $\text{ls-send}_p(S, r)$
4: for all $(m, q) \in S$ do
5: $a$-send$_p((r, m), q)$
6: for all $q' \notin \{q''|(m', q'') \in S\}$ do
7: $a$-send$_p((r, \varepsilon), q')$

8: operation $\text{ls-receive}_p(r)$
9: repeat
10: $\text{undelivered}_p \leftarrow \text{undelivered}_p \cup \{a$-receive$_p()\}$ /* skip if $\varepsilon$ */
11: $\text{suspect} \leftarrow \text{a-queryFD}_p()$
12: until $\{q | (r, *, q) \in \text{undelivered}_p \cup \text{suspect} = \Pi$
13: $\text{del} \leftarrow \{(r', *, *) \in \text{undelivered} | r' \leq r\}$
14: $\text{undelivered}_p \leftarrow \text{undelivered}_p \setminus \text{del}$
15: return del

Lemma 1. An arbitrary valid algorithm $A$ for $\diamondsuit$LS whose operations are implemented according to Algorithm 1 is a valid algorithm for Async+\diamondsuitP.

Proof. An execution for $\diamondsuit$LS, consisting of a sequence of instantiations of $\text{ls-send} \rightarrow \text{ls-receive} \rightarrow \text{ls-send} \rightarrow \text{ls-receive} \rightarrow \ldots$. A Step in Async+\diamondsuitP consists of at least one of the operation such that any sequence of Async+\diamondsuitP operations lead to a valid algorithm and our lemma follows trivially.

Lemma 2. No correct process, executing Algorithm 1 can block forever in some round $r$.

Proof. Processes cannot block during the $\text{ls-send}$ operation. Processes can block only in line 12. Assume by contradiction that there is a minimal round $r$ in which some correct process $p$ blocks forever. Since $r$ is the smallest round in which a correct process blocks forever, all correct processes must eventually reach line 12 for round $r$ and thus — by the assumption of valid $\diamondsuit$LS executions — must have executed either line 5 or line 7 and thus have sent round $r$ messages to all. Since, after the failure detector reaches strong completeness and strong accuracy, and after the last process has crashed, $p$ eventually just waits for messages sent by correct processes. Since, by Reliability these messages must eventually be received by $p$, $p$ must unblock, which is the required contradiction.

Lemma 3. An arbitrary valid algorithm $A$ for $\diamondsuit$LS whose operations are implemented according to Algorithm 1 fulfills Reliability, Integrity and No Duplication.
Algorithm 2 Reducing ◦Synch to ◦LS, code for a process \( p \).

1: variables
2: \( r \in \mathbb{N} \), initially 0
3: \( \text{buffer}_p \subseteq \mathbb{N} \times M \times \Pi \), initially \( \epsilon \)
4: operation \( s\text{-send}_p(m, q) \)
5: \( \text{ls\text{-send}}_p(\{(m, q)\}, r) \)
6: \( \text{buffer}_p \leftarrow \text{buffer}_p \cup \text{ls\text{-receive}}_p(r) \)
7: \( r \leftarrow r + 1 \)
8: operation \( s\text{-receive}_p() \)
9: \( \text{ls\text{-send}}_p(\emptyset, r) \)
10: \( \text{buffer}_p \leftarrow \text{buffer}_p \cup \text{ls\text{-receive}}_p(r) \)
11: \( r \leftarrow r + 1 \)
12: \text{return buffer}_p

Proof. Reliability. For every \( p \in \Pi \), every time \( \text{ls\text{-send}} \) is called, \( a\text{-send} \) is invoked as many times as necessary to send the respective the message to the respective processes. Besides, every time \( \text{ls\text{-receive}} \) is called, \( a\text{-receive} \) is invoked until all messages for the current round are received. After that all messages for rounds less than or equal to the current round are delivered. By Lemma 2 progress is ensured such that eventually every message is delivered due to Reliability of ◦LS.

Integrity and No Duplication. Follows from the fact that a message is only delivered by \( \text{ls\text{-receive}} \), iff it was delivered by \( a\text{-receive} \), and the fact that \( \text{Async} + \diamond \text{P} \) fulfills both Integrity and No Duplication.

Lemma 4. An arbitrary valid algorithm \( \mathcal{A} \) for ◦LS whose operations are implemented according to Algorithm 1 fulfills Eventual Lockstep.

Proof. We have to ensure that eventually all messages are delivered in the round they were sent.

Processes \( p \) deliver messages only in line 15, i.e., after unblocking in line 12 which ensures that after the failure detector reaches strong completeness and strong accuracy, and after the last process has crashed and all its messages are delivered, only the messages of all correct processes for the current round are delivered.

By the simulation in Algorithm 1 and Lemmata 1, 3, and 4 we get:

Theorem 1. ◦LS is reducible to Async+◊P.

3.2 Reducing Eventual Synchrony to Eventual Lockstep

In this section we show that an arbitrary algorithm \( \mathcal{A} \) for ◦Synch can correctly be executed in an ◦LS system model, by giving implementations of the ◦Synch operations \( s\text{-send} \) and \( s\text{-receive} \).
Lemma 5. An arbitrary valid algorithm $A$ for $\diamond Synch$ whose operations are implemented according to Algorithm 2 is a valid algorithm for $\diamond LS$.

Proof. For $\diamond LS$ we just have to ensure that an execution consists of a sequence $ls-send \rightarrow ls-receive \rightarrow ls-send \rightarrow ls-receive \rightarrow \ldots$. Both operations of Algorithm 2 first call $ls-send$ and then $ls-receive$ such that an arbitrary sequence of $s-receive$ and $s-send$ is a valid $\diamond LS$ execution. \hfill \square

Lemma 6. An arbitrary valid algorithm $A$ for $\diamond Synch$ whose operations are implemented according to Algorithm 2 fulfills Reliability, Integrity and No Duplication.

Proof. Follows from Reliability, Integrity and No Duplication of $\diamond LS$. \hfill \square

Lemma 7. An arbitrary valid algorithm $A$ for $\diamond Synch$ whose operations are implemented according to Algorithm 2 fulfills Eventual Synchrony with $\Delta = 0$ and Synchronous Processors with $\Phi = 1$.

Proof. Eventual Synchronous Processes. There is eventually a one-to-one mapping of the instances of $s-send$ and $s-receive$ operations to lock step rounds—i.e., each $\diamond Synch$ operation corresponds to two $\diamond LS$ operations. Eventually, if a message $m$ is sent via an $s-send$ corresponding to round $r$ at the sender, it is delivered via the first $s-receive$ operation for some round $r' \geq r$. By the definition of $\diamond Synch$ we get $\Delta = 0$.

Synchronous Processors. In every lock step round either an $s-send$ or an $s-receive$ operation must be performed at a correct process. That is, in every lock step round every correct process performs an operation which, by the definition of $\diamond Synch$, corresponds to $\Phi = 1$. \hfill \square

By the simulation in Algorithm 2 and Lemmata 5, 6, and 7 we get:

Theorem 2. $\diamond Synch$ is reducible to $\diamond LS$.

3.3 Reducing Asynchrony with $\diamond P$ to Partial Synchrony

In this section we review the results of Chandra and Toueg [1], in the sense that we use their failure detector implementation for implementing the $a$-query$FD$ operation. For sake of self-containment of our results, we put their algorithm and proof in our framework.

We thus show that an arbitrary algorithm $A$ for $Async+\diamond P$ can correctly be executed in an $ParSync$ system model, by giving implementations of the $ParSync$ operations $a-sync$ and $s-receive$.

Lemma 8. An arbitrary valid algorithm $A$ for $Async+\diamond P$ whose operations are implemented according to Algorithm 3 is a valid algorithm for $ParSync$. 

8
**Algorithm 3** Reducing Async+ to ParSync, code for a process $p$ [1].

1: **variables**
2: $\text{suspect}_p \subseteq \Pi$, initially empty
3: $\forall q: \text{timer}_p[q], \text{delta}_p[q] \in \mathbb{N}$, initially 0
4: $\text{buffer}_p$: FIFO queue of tuples $(m, q)$, where $m \in M, q \in \Pi$, initially empty

5: **function** $\text{update}()$
6: $S \leftarrow \text{s-receive}_p()$
7: **for all** $(m, q) \in S$ **do**
8: \hspace{1em} if $m \neq \epsilon$ **then**
9: \hspace{2em} enqueue $(m, q)$ in $\text{buffer}_p$
10: \hspace{2em} $\text{timer}_p[q] \leftarrow 0$
11: \hspace{1em} if $q \in \text{suspect}_p$ **then**
12: \hspace{2em} $\text{suspect}_p \leftarrow \text{suspect}_p \setminus \{q\}$
13: \hspace{2em} $\text{delta}_p[q] \leftarrow \text{delta}_p[q] + 1$
14: **for all** $q$, where $\text{timer}_p[q] > \text{delta}_p[q]$ **do**
15: \hspace{2em} $\text{suspect}_p \leftarrow \text{suspect}_p \cup \{q\}$

16: **operation** $\text{a-receive}_p()$
17: $\text{update}()$
18: remove one $(m, q)$ from $\text{buffer}_p$ and **return** $(m, q)$ [or $\epsilon$ if $\text{buffer}_p$ is empty]

19: **operation** $\text{a-queryFD}_p()$
20: $\text{update}()$
21: **return** $\text{suspect}_p$

22: **operation** $\text{a-send}_p(m, q)$
23: $\text{s-send}_p(m, q)$
24: $\text{delta}_p[q] \leftarrow \text{delta}_p[q] + 1$
25: $\text{update}()$

---

**Proof.** For ParSync we just have to ensure that an execution consists of an arbitrary sequence of $\text{s-send}$ and $\text{s-receive}$ operations. Since this are the only operations that are used in Algorithm 3 we are done. \hfill $\Box$

**Lemma 9.** An arbitrary valid algorithm $A$ for Async+ whose operations are implemented according to Algorithm 3 fulfills Reliability, Integrity and No Duplication.

**Proof.** Integrity follows trivially from Integrity of ParSync. No Duplication follows from No Duplication of ParSync and the fact that a message is removed from $\text{buffer}_p$ when it is delivered to $p$. Reliability follows from Reliability of ParSync and the FIFO property of $\text{buffer}_p$. \hfill $\Box$

**Lemma 10.** An arbitrary valid algorithm $A$ for Async+ whose operations are implemented according to Algorithm 3 fulfills Strong Completeness and Eventual Strong Accuracy.
Proof. Strong Completeness. If a process crashes, there is a time after which no more messages from that process will be received. Thus, if the process does not already belong to $\text{suspect}_p$, it will when $\text{timer}_p[q] > \text{delta}_p[q]$ and remain so, as there will be no more messages from it being received.

Eventual Strong Accuracy. There is a time after which for all correct processes every message sent by $a$-send through $s$-send will arrive in the next $s$-receive (as $\Delta = 0$ and $\Phi = 1$). As $\text{timer}_p[q]$ is always increased between $s$-send and $s$-receive, and turns zero when a new message arrives, $\text{timer}_p[q]$ will remain as zero for every correct process $q$.

By the simulation in Algorithm 3 and Lemmata 8, 9, and 10 we get:

**Theorem 3.** Async+$\diamondsuit$P is reducible to ParSync.

Thus, we can conclude that all discussed models are equivalent in terms of solvability:

**Corollary 1.** ParSync $\preceq$ Synch $\preceq$ LS $\preceq$ Async+$\diamondsuit$P $\preceq$ ParSync.

4 Discussion

In this paper we have shown that asynchrony augmented with an eventually perfect failure detector is equivalent in terms of solvability to partial synchrony, eventual synchrony and eventual lockstep.

The main observation in the relation to [2] is concerned with the term "eventually": In the asynchronous model augmented with $\diamondsuit\mathcal{P}$, two things happen in every execution: (1) eventually, the failure detector becomes accurate, and (2) eventually, the last process crashes and all its messages are received.

The model considered in [2] (asynchrony with $\mathcal{P}$), however shares only (2) while the failure detector considered satisfies perpetual strong accuracy. It was shown in [2] that given (2), even perpetual strong accuracy — and thus $\mathcal{P}$ — is too weak to implement a model where every round is communication closed [6], as is required by the synchronous model of computation: If a process $p$ crashes, $\mathcal{P}$ does not provide information on the fact whether there are still messages sent by $p$ in transit (cf. [7]).

For showing our equivalence result, we were interested in ensuring communication closed rounds just eventually. We observe that (1) and (2) are sufficient to achieve this. After (1) and (2), all processes are correct, they will never be suspected and thus all their messages are received in the same round as they were sent. Thus we achieve communication closed rounds eventually which is equivalent to eventual lock-step and thus eventual synchrony.
References


