On the Possibility of Consensus in Asynchronous Systems with Finite Average Response Times

Christof FETZER
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932, USA
cristof@research.att.com

Ulrich SCHMID
TU Wien
Treitlstraße 3
A-1040 Vienna, Austria
s@ecs.tuwien.ac.at

Abstract

It has long been known that the consensus problem cannot be solved deterministically in completely asynchronous distributed systems, i.e., systems (1) without assumptions on communication delays and relative speed of processes and (2) without access to real-time clocks. In this paper we define a new asynchronous system model: Instead of assuming reliable channels with finite transmission delays, we assume stubborn channels with a finite average response time (if neither the sender nor the receiver crashes), and we assume that there exists some unknown physical bound on how fast an integer can be incremented. Note that there is no limit on how slow a program can be executed or how fast other statements can be executed. Also, there exists no upper or lower bound on the transmission delay of messages or the relative speed of processes. The are no additional assumptions about clocks, failure detectors, etc. that would aid in solving consensus either. We show that consensus can nevertheless be solved deterministically in this asynchronous system model.

Keywords: impossibility, consensus, asynchronous systems, eventually perfect failure detector.

1 Introduction

The consensus problem is a fundamental problem in the area of distributed computing and has therefore been very thoroughly investigated over the last two decades (e.g., see [17, 10, 13, 12, 27]). The consensus problem is defined as follows:

(Agreement) All processes that decide, decide on the same value.

(Validity) The value a process decides is the initial value of some process.

(Termination) Every correct process eventually decides.

It has been shown in [17] that one cannot deterministically solve the consensus problem in completely asynchronous systems, which are characterized by the following properties (see [17], p.375):

- no assumptions on the relative speed of processes,
- no assumptions on the delay time of delivering messages, and
- no synchronized or bounded-drift clocks.

In this paper we introduce a new asynchronous system model (we call it the Finite Average model or FA model) that

- does not bound the relative speed of processes or minimum speed of processes,
- does not postulate upper or lower bounds on the messages delivery times,
- does not assume that the system stabilizes, and
- does not assume clocks, failure detectors, or other extensions of the model.
Rather, it assumes unknown finite average response time and unknown finite maximum speed for incrementing an integer. Despite of those weak assumptions, the FA model permits a deterministic solution of the consensus problem.

The FA model makes the following non-standard assumptions:

- Incrementing an integer takes some unknown time \( \geq G > 0 \).
- The system enforces some basic flow control. It only guarantees to deliver a message to a correct process \( q \) if the process \( p \) that sent the message waits for an acknowledgment before sending the next message.
- The average response time is finite, i.e., the average time until the acknowledgment of a message sent between two correct processes arrives is finite.

The FA model is justified by the following observations:

- The speed of incrementing an integer cannot become infinitely fast (even if we would constantly be updating the underlying hardware). Eventually, the speed of incrementing an integer will reach some physical or economical limit (e.g., speed of light and lower bounds on the area/volume needed to store a bit) that will prevent us from further increasing the speed of incrementing an integer.
- In systems that do not use any flow control mechanisms processes can exhibit longer and longer response times because “fast” processes can overload “slower” processes.
- If we enforce some basic flow control, the response times of a correct process pair is typically not continuously increasing. The response times might go up during times when the link between two correct processes needs to be repaired. However, eventually links are repaired and response times might drop again.

We show in this paper that one can solve consensus deterministically in the FA model. Instead of describing a consensus protocol, we show how to implement an eventually perfect failure detector [2]. Our failure detector can be used with an already published consensus protocol [19] that works with stubborn channels to solve the consensus problem.

An eventually perfect failure detector has the following properties:

- **Completeness:** Eventually, all correct processes suspect all crashed processes.
- **Eventual Strong Accuracy:** Eventually, no correct process suspects any correct process.

Our eventually perfect failure detector (EA-FD) is based on the following idea. We can continuously increment a counter to establish a very weak notion of the passage of time. This notion of time is sufficient to implement a clock that provides a “subjective” notion of slow and fast messages: The acknowledgment of a slow message \( m \) arrives after the timeout for \( m \) expired and the acknowledgment of a fast message \( n \) arrives before the timeout for \( n \) expires. The timeout for messages is dynamically adapted according to the classification of earlier messages. However, unlike most other timeout-based failure detectors a process increases the time-out when it receives a fast message but might decrease the timeout when it receives a slow message. We show that this strategy ensures that the failure detector will eventually be perfect.

**Advantages:** The commonly employed partially synchronous system model of [7] is not a panacea for solving consensus: Assuming that the response times in large scale systems are eventually bounded is questionable in presence of transient link failures, for example. In addition, very large systems might never stabilize completely. Ensuring and/or justifying such “global” assumptions in such systems is in fact intrinsically difficult. By contrast, in the FA model, we only make the “local” assumption that the average response time for messages sent between a pair of correct processes is finite. This is much easier to enforce and/or justify; we typically only have to make sure that failed links are eventually repaired.
The advantage of our failure detector is that given the assumptions of the FA model are true, one can prove that algorithms like the consensus algorithm of [19] are correct and in particular, will always terminate. However, in real systems one does not only want correctness but also performance. We show how the EA-FD failure detector can be “fused” with other timeout-based failure detectors to increase the chances of an earlier termination without sacrificing the guaranteed termination.

**Outline:** We introduce the FA model in Section 2 and describe a new eventually perfect failure detector EA-FD in Section 3. We discuss some practical aspects like combining EA-FD with an adaptive but not necessarily eventually perfect failure detector in Section 4. Section 5 discusses related work and Section 6 concludes the paper. Optional material can be found in the Appendix.

## 2 The Finite Average Model

This section presents our new asynchronous system model (FA model). In this model there are no assumptions on the transmission delay of messages, on the relative speeds of processes, and there are no additional entities like clocks or failure detectors.

Instead of using reliable channels, our system model is based on fixed-size stubborn channels which are a slight variant of the stubborn channels introduced in [19]. Note that stubborn channels do not strengthen our model in comparison to the FLP model[3]. The main advantages of stubborn channels are that one only needs a bounded amount of memory to implement them atop of unreliable channels, and that they bound the number of messages that can be in transit.

Before we can define the FA model in Section 2.2, we need to introduce a few basic definitions in Section 2.1. Note that while we use real time in the definition of the FA model, none of the processes of a FA system has access to a real time clock.

### 2.1 Definitions

An execution of a protocol consists of a sequence of actions. To simplify the paper, we do not introduce a formal notion of a run. Note however that some definitions (like correct<sub>p</sub>) are defined with respect to an implicitly given run.

**Definition 1 (correct).** We define that predicate correct<sub>p</sub> is true iff p does not crash in the entire execution.

**Definition 2 (acknowledged stubborn channel).** If a correct process p sends a message m of size ≤ S to a correct process q via an acknowledged stubborn channel and p delays sending any other message to q until it receives an acknowledgement for m, then eventually m will be delivered to q and p will receive an acknowledgement that q has delivered m.

**Definition 3 (fixed-size).** For S < ∞, we call these stubborn channels fixed-size because only messages up to size S can be transmitted.

We assume that the receiver q of a message m can piggyback a message of a size up to S on the acknowledgment message of m. We denote the fixed-size acknowledged stubborn channel from p to q by SC<sub>p→q</sub>.

**Operational Aspects.** Other than the poll-based model defined in [17], we define a push-based model: The arrival of a message automatically triggers the execution of an action. Actions are executed in sequence and hence, the delivery of messages to a process is sequentialized. Operationally, we define the following interface for fixed-size stubborn channels (see Figure 1):

- To send a message of size ≤ S, a process can call a primitive send. By expression “send<sub>sc</sub>(m)” to q” we denote that a process sends a message m to process q via the fixed-size stubborn channel. The size of m, denoted by size(m), must be at most S, i.e., size(m) ≤ S.

- A message m from p to q is delivered by an action on q. In the pseudocode we express this delivery action as follows: “on sc_deliver m from p { ... sc_piggyback(n) to p; ... }”. After executing this action, q sends an acknowledgment to q. The acknowledgment lets p know that it can send the next message. q can piggyback a message n on the acknowledgment message. However, the size of n has to be bounded by S, i.e., size(n) ≤ S.

---

3. We show in Section A.2 that stubborn channels can be implemented using reliable channels.

4. We permit actions to be nested in the sense that an action can execute another action in very much the same way an action would call a function. Actions itself and nested actions must be executed in an uninterruptable fashion, however, i.e., must not be preempted by another message arrival.
• The acknowledgment together with the optional piggybacked message \( n \) is delivered by an action on \( p \). In the pseudocode we express this action as follows: 

\[
\text{on sc_ack}(n) \text{ from } q \{ \ldots \}
\]

Note that the message \( n \) that was piggybacked on the acknowledgment message is not acknowledged.

\[\text{sc_send}(m) \text{ to } q \]

\[\text{vs_piggyback}(n) \text{ to } p;\]

\[\text{on sc_deliver } m \text{ from } p\]

Figure 1: Communication is push based: The arrival of a message automatically triggers the execution of an action. The reception and processing of a message is acknowledged. We permit application messages to be piggybacked on an acknowledgment message. Note however that these piggybacked messages are not acknowledged.

**Response Time.** The FA model assumes that the average response time is finite. The response time of a message \( m \) is the real-time duration between the time a message \( m \) is sent and the time the acknowledgment for \( m \) arrives. In Figure 1 the response time of message \( m \) is the real-time duration between point \( A \) and \( C \).

To make our assumption as weak as possible, the FA model will not even guarantee a finite average response if a process does not wait for an acknowledgment before sending the next message via the same link. To formalize this, we first assign certain messages sent via a channel \( SC_{p \rightarrow q} \) a unique natural number.

**Definition 4 (message enumeration).** We enumerate (starting at number 1) exactly those messages sent via a fixed-size acknowledged stubborn channel \( SC_{p \rightarrow q} \) that are acknowledged and are acknowledged before the next message is sent via \( SC_{p \rightarrow q} \). Set \( I_{p \rightarrow q} \) contains this enumeration for link \( SC_{p \rightarrow q} \).

To explain this definition, consider that a process \( p \) always waits for an acknowledgment from \( q \) before sending a message via \( SC_{p \rightarrow q} \) and all messages are acknowledged. In this case, this definition assigns the first message sent via \( SC_{p \rightarrow q} \) the number 1, and the next message the number 2, etc. However, if process \( p \) sends a new message \( n \) via \( SC_{p \rightarrow q} \) before it received an acknowledgment for the previous message \( m \) that \( p \) sent via \( SC_{p \rightarrow q} \), then the definition does not assign a number to \( m \) (see Figure 2). Also if a message \( m \) is never acknowledged, the definition does not assign a number to \( m \). In particular, messages sent to a crashed process are never acknowledged and hence, these are not enumerated.

**Definition 5 (response time).** For \( i \in I_{p \rightarrow q} \), \( RT_{p \rightarrow q}(i) \) is the duration between the point in real time the \( i \)-th message is sent by \( p \) to \( q \) via \( SC_{p \rightarrow q} \) and the point in real time at which \( p \) receives the acknowledgment for \( m_i \).

Note that the response time cannot be measured by the processes since they do not have a real-time clock.

### 2.2 FA Model

A system consists of \( N < \infty \) processes.

- **(A0)** A process executes actions in sequence. A process \( p \) can prematurely stop executing its actions if \( p \) crashes. An action can send multiple messages but at most one message per destination process. Actions are not crash atomic, i.e., a process can crash during the execution of an action.

- **(A1)** Each pair of processes \( p, q \) can communicate via a fixed-size acknowledged stubborn channel \( C_{p \rightarrow q} \) of size \( S \).

- **(A2)** The average response time converges\(^5\) and it is
finite:
\[
\forall p \forall q : \lim_{i \to |I_p \to q|} \frac{\sum_{1 \leq j \leq i} RT_{p \to q}(j)}{i} < \infty
\]

(A3) Incrementing an integer number by 1 takes an unknown minimum amount of time \( G > 0 \).

(A4) Processes can only fail by crashing but at most a minority of the processes can fail. Crashed processes do not recover.

The maximum response time and maximum message transmission delay is unbounded in the FA model, and so is the ratio of the fastest and slowest process and the ratio of the fastest and slowest message delay. The model does not assume or specify real-time clocks.

2.3 Remarks

Assumption (A3) can be motivated by fundamental limits on how fast a processor can perform an addition by 1. For example, the bounded speed of light and limits on how dense bits can be packed, limit the speed with which two numbers can be added. To make (A3) as weak as possible, we do not limit the maximum speed of any other statements. Note that we chose the increment operation for convenience: The existence of any operation for which there exists an unknown but positive minimum execution time would be sufficient.

We use fixed-size acknowledged stubborn channels (A1) instead of reliable channels to address the following issue: A fast process \( p \) that sends messages to a slower process \( q \) without flow control via a reliable channel can produce an unbounded number of messages that are in transit to \( q \). When the number of messages in transit can be unbounded, a finite average response time (like postulated in A2) is not necessarily valid. A stubborn channel can limit the number of messages in transit, i.e., it can enforce a very strict flow control.

By definition, \( I_p \to q \) only contains messages that are acknowledged. Hence, if the number of messages sent via a channel \( SC_p \to q \) is finite (i.e., \( |I_p \to q| < \infty \)), the average response time is always finite. The FA model states that even if a process sends infinitely many messages via a channel, the average response time stays finite. This is a reasonable assumption for fixed-size acknowledged stubborn channels as long as broken network links are repaired and as long as physical network links and processors are not getting slower and slower.

The FLP model requires that messages are eventually delivered, i.e., that the transmission delay is finite (this also implies that the response time is finite). In the FA model, we have the additional assumption that the average response time is also finite. For example, the FLP model permits that the transmission delay of any subsequence of the messages sent via a reliable channel can be monotonically increasing, e.g., 1, 2, 3, . . . . This is not allowed in the FA model due to (A2). Note however that such a behavior is allowed for infinite subsequences of messages sent via a channel as long as there are sufficiently many faster messages sent via the same channel that compensate for the increasingly slow messages, thereby, ensuring that the average stays finite. For example, the response time of all messages sent via a channel might be 1, 2, 1, 3, 1, 1, 4, 1, 1, 1, 5, . . . .

Note that in systems in which the response times are bounded, the average response time converges and is finite. As we have demonstrated, a finite average response time does not mean that response times are bounded.

One has to expect that the transmission delay and hence the response time of a message increases with the size of the message. If a process \( p \) could send messages of unbounded size, \( p \) could gradually increase the response time. To address this issue, we permit only messages that are of size \( \leq S \).

3 Implementation

We show how to implement an eventually perfect failure detector \( EA-FD \) in the FA model (Section 3.2). This failure detector is timeout-based and thus needs a clock. We show in Section 3.1 how to implement a clock with very weak properties on top of the FA model that is still sufficiently strong to implement the failure detector. In addition, consensus protocols need to send variable size messages. (Section A.1 in the Appendix shows how to implement variable-size stubborn channels on top of the failure detector layer of Section A.1.)

3.1 Weak Clock

The FA model does not contain clocks. In particular, processes cannot accurately measure the duration of intervals. However, for each process \( p \), one can implement a monotonic clock \( W_p \) with very weak seman-
tics (see Figure 3):
\[(W) \forall p, \forall s, \forall t : s < t \land \text{correct}_p \Rightarrow 0 \leq W_p^t - W_p^s \leq \lceil \frac{t-s}{G} \rceil\]
where \(W_p^t\) is the value of clock \(W_p\) at real-time \(t\) and \(G\) is the unknown minimum time to increment an integer from assumption (A3). Note that since \(G\) is not known, one does not know how fast the clock can proceed. In addition, there is no limit on how slow the clock proceeds.

A correct process will execute action \(\text{Tick}\) infinitely often. Hence, we can guarantee that the clock of a correct process is unbounded
\[(M) \forall p, \forall B : \text{correct}_p \Rightarrow \exists t : W_p^t > B\]

Note that the relative speed of two clocks is unbounded because there is no bound on how slowly a clock can proceed.

Figure 3: Weak Clock Implementation. Action Tick can always be executed (i.e., no precondition) and a correct process will always eventually execute Tick.

Correctness Argument. Assumption (A3) implies that the execution of function \(\text{tick}\) takes at least some unknown time \(G > 0\) because the function increments variable \(c\) (line 3). Variable \(c\) is only modified by function \(\text{tick}\) and assumption (A0) implies that action \(\text{Tick}\) and hence function \(\text{tick}\) is always called sequentially. Therefore, variable \(c\) can be incremented at most every \(G\) time units. This implies property (W). Note that the ceiling function in property (W) is needed since term \(W^t\) is defined for all times and in particular, for times just before and just after \(c\)’s value changes.

Since a correct process \(p\) will execute action \(\text{Tick}\) infinitely often, for each \(B\) we can find a time \(t\) such that \(p\) has executed \(\text{Tick}\) at least \(B\) times by time \(t\). In other words, property (M) holds.

3.2 Eventually Perfect Failure Detector \(\text{EA-FD}\)

Our eventually perfect failure detector \(\text{EA-FD}\) (see Figure 4) is based on the following idea: \(\text{EA-FD}\) measures the response time of messages sent by the application using the weak clock \(W\). It maintains a timeout value that is used to classify messages as either “fast” or “slow”. Note that a classification of a message is not predetermined because the timeout changes over time and the speed of the clock can be very variable.

For each channel \(SC_{p \rightarrow q}\) there is at most one message \(m\) for which \(p\) is waiting for an acknowledgment from \(q\). Function \(\text{is}_\text{suspected}(\text{q})\) returns true if the timeout for the current unacknowledged message has already expired. If there is no unacknowledged message, the function sends one. If process \(q\) has crashed and process \(p\) continues to query the status of \(q\) by calling \(\text{is}_\text{suspected}(\text{q})\), eventually the timeout will expire for some message to \(q\) because \(q\) will not acknowledge any message after its crash.

The protocol counts the number of slow messages and the number of fast messages between two consecutive slow messages. There is a timeout per link which is only updated at the arrival of an acknowledgment via that link. The timeout increases logarithmically with the total number of slow messages and linearly with the number of fast messages since the last slow message. Whenever a slow message arrives, the number of fast messages is set to zero and therefore results in a drop of the timeout (if the number of fast messages was greater than zero). Note that each wrong suspicion (i.e., a non-crashed process is wrongly suspected to have crashed) is caused by a slow message. The intuition behind this timeout scheme is that the average response time of a link increases slowly with the number of slow messages sent via this link. In particular, if the number of wrong suspicions were infinite, the average response time of the link would be infinite too. Since the average response time is finite, there cannot be infinitely many wrong suspicions. Since there are only a finite number of links, the failure detector will eventually be accurate.

Let us come back to the completeness property. Even though the timeout of a link \(SC_{p \rightarrow q}\) might grow over time, the timeout is always finite and the timeout does not change while there is an unacknowledged message. This makes sure that if a message is not acknowledged because the remote process \(q\) has crashed, eventually a correct \(p\) will suspect \(q\) since the timeout will eventually expire due to property (M) of the weak clock.

Theorem 1 (Eventually Perfect). Failure detector \(\text{EA-FD}\) is an eventually perfect failure detector.

Proof. To show that EA is complete, we have to show the following: If a correct process \(q\) calls \(\text{is}_\text{suspected}(p)\)
an unbounded number of times for a crashed process $p$, eventu-
al $is\_suspected(p)$ always returns true. Let us assume that process $p$ crashes and process $q$ queries the status of $p$ an unbounded number of times. Hence, there exists an infinite sequence $A_1, A_2, \ldots$ of actions which (1) all happened after the crash of $p$, (2) all acknowledgments from $p$ to $q$ have been delivered, (3) $q$ executes $A_i$ before $A_{i+1}$, and (4) in each action $A_i$ process $q$ calls $is\_suspected(p)$. If there exists no un-
alsted message that $q$ has sent to $p$ before executing $A_1$, $A_1$ sends a message to $p$ which will never be acknowledged since $p$ has already crashed. Hence, no later than action $A_2$ variables $unacked[p]$ (contains sent time of last unacknowledged message from $q$ to $p$ and 0 otherwise), $numFast[p]$ (contains the number of fast messages from $q$ to $p$ since last slow message from $q$ to $p$), and $slowmsgs[p]$ (contains the total number of slow messages from $q$ to $p$) will not change anymore. Because of property (M) of $q$’s clock $W$, there exists an $i$ such that
\[
unacked[p]+(1+numFast[p])*(1+log(1+slowmsgs[p]))<W()
\]
which means that in all actions $A_{j,j\geq1}$ process $p$ is sus-
ppected by $q$.

To show that EA is eventually accurate, we need to show that eventually no correct process will ever be suspected. We show this by contradiction and hence assume that there exists a correct process $p$ that sus-
pcts a correct process $q$ infinitely often. A wrong sus-
picion of $q$ is caused by a slow message. Therefore, each wrong suspicion of $q$ is eventually corrected by the arrival of an acknowledgment from $q$.

We derive a lower bound for the average response time as follows: we use 0 as the lower bound of all fast messages and use the current timeout value as a lower bound for a slow message. Due to our selec-
tion of $p$ and $q$, we know that $|I_{p\rightarrow q}| = \infty$ and the number of slow messages is infinite. Let $numFast_q^j$ be the number of fast messages between the $j$-th and $j+1$-th slow message from $p$ to $q$. The total number of messages from $p$ to $q$ at the arrival of the ac-
nowledgment of the $k$-th slow message is therefore $\sum_{0\leq j \leq k}(1 + numFast_q^j)$. The lower bound for the $j$-
th slow message is $G(1 + numFast_q^j)(1 + \log(j + 1))$. Hence, we know that:
\[
\lim_{i \rightarrow -\infty} |I_{q\rightarrow p}| \sum_{1 \leq j \leq i} RT_q(p,j) \geq \\
lim_{k \rightarrow \infty} \frac{G \sum_{0 \leq j \leq k}(1 + numFast_q^j)(1 + \log(j + 1))}{\sum_{0 \leq j \leq k}(1 + numFast_q^j)} = \infty.
\]
This is a contradiction to assumption (A2).

4 Practical Aspects

Although the main contribution of our work is to show that one can solve consensus in the FA model, we also address some practical aspects of our approach.
They are derived from our C++ prototype implementation, which reveals that algorithms can readily be used for building up real systems.

4.1 Clock Implementation

Most computers have a hardware clock. For example, Intel Pentium processors have a counter (tsc) that is incremented during each CPU cycle. Hence, the speed of this counter increases with higher CPU clock frequencies. It is nevertheless reasonable to use this counter as an implementation of the weak clock introduced in Section 3.1. The actual speed of the counter in a run does not matter as long as it is not approaching infinite (which would be physically impossible). The problem of counter wrapping can be defeated by using unbounded integers, as e.g. provided by a GNU MP library.

4.2 Failure Detector Fusion

The timeouts of failure detector EA-FD can grow quite large over time. For algorithms like a consensus algorithm that typically run only for a few rounds before terminating, one can decrease the expected time for termination without sacrificing the guaranteed termination by fusing EA-FD with an adaptive timeout-based failure detector that adjusts its timeouts according to the current system behavior (e.g., [16, 4, 8]) but which does not guarantee termination in the FA model.

Typically, a timeout-based failure detector will set its timeout based on the predicted current response time which might be based on criteria like the average response time, the average response time over the last $K$ messages, the last observed response time, the maximum observed response time, etc. We fuse such a failure detector QOS-FD with EA-FD by using QOS-FD to compute the timeouts for a link $SC_{p \rightarrow q}$ until the number of slow message on $SC_{p \rightarrow q}$ reached a given threshold. After that, the timeouts will be computed by EA-FD which ensures that the failure detector will eventually be perfect (see Figure 5).

4.3 k-Stubborn Channels

If one needs to optimize the throughput of a channel, it does not make sense to permit only one unacknowledged message per channel. Our approach can easily be extended to k-stubborn channels [19], i.e.,

```cpp
const int FUSION_THRESHOLD;

function fd_timeout (process p) {  
  if (slowmsgs[p] < FUSION_THRESHOLD) {  
    return qos_fd_timeout ();  
  } else {  
    return (1+numFast[p])*(1+log(1+slowmsgs[q]));  
  } }
```

Figure 5: Failure Detector Fusion. We replace the fd_timeout function of Figure 4 by a new function that uses the timeout of QOS-FD until the total number of slow messages of a link reaches FUSION_THRESHOLD. From then on, we use the timeout of EA-FD which guarantees that eventually there will be no wrong suspicions.

channels in which up to $k < \infty$ instead of only 1 message can be in transit.

5 Related Work

The impossibility of deterministic consensus in the FLP model [17] stimulated a wealth of research. In [10], the exact borderline between models where consensus can/cannot be solved has been determined for 5 key aspects (communication delays, speed ratio, message order, broadcast, atomicity). In particular, consensus cannot be solved in systems where either processing or communication delays are unbounded.

An important class of models that allow consensus to be solved are known as partially synchronous models. The seminal paper [13] classifies partial synchrony according to whether bounds upon the maximum relative processing speeds and the maximum absolute communication delays exist but are either unknown, or are known but hold only after some unknown global stabilization time GST. Those two models were combined into a single generalized partially synchronous model in [7].

Alternative models have been proposed, which augment asynchronous systems with additional facilities and/or properties. The most prominent example are unreliable failure detectors, introduced in [6], which add an oracle that provides processes with hints about crashed processes. [5] provides the weakest failure detector for solving consensus, and [7] contains a comprehensive study of all classes of failure detectors that are sufficiently strong for solving consensus.

Failure detectors are specified in an abstract axiomatic way, however, which raises the question of how to implement them in a real system: Due to the consensus
impossibility in the FLP model [17], no sufficiently strong failure detector can be implemented in purely asynchronous systems. Stronger models are hence required when implementing a failure detector.

Most implementations of eventual-type failure detectors [7] [22] [13] [23] [24] [11] [1] [25] [8] [3] [20] [16] [21] [4] [2] [28] rely upon the generalized GST model of [7], or even a synchronous model. The simple implementation of an eventually perfect failure detector $\Diamond P$ in [7] is based upon monitoring periodic “I am alive”-messages using adaptive (increasing) timeouts at all receiver processes: Starting from an a priori given initial value, the timeout value is increased every time a false suspicion is detected. By restricting the recipients of “I am alive”-messages from all processors to suitably chosen subsets, a less costly implementation of an eventually strong failure detector $\Diamond S$ was derived in [25]. Alternative FD implementations, which use polling by means of ping/reply round-trips instead of “I am alive”-messages, have also been proposed. The message-efficient algorithms of [24] use a logical ring, where processors poll only their neighbors and use an adaptive (increasing) timeout for generating suspicions.

Other instances of timeout-based failure detectors are the hardware watchdogs permitting a timely crashing of processes proposed in [15], which allow to solve consensus in the timed asynchronous model [9], and the fast failure detectors of [21], which use head-of-the-line-scheduled FD-level messages to speed up detection time. Note that not all existing adaptive timeout approaches can decrease the timeout value, cp. [25] [16], i.e., cannot adapt to (slowly) varying delays over time.

There are alternatives to timeout-based failure detector implementations, which usually require an asynchronous model plus some additional assumptions. For example, the implementation of the leader oracle $\Omega$ in [2], which outputs just a single—eventually common—process that is considered up and running, assumes a partially synchronous systems where only a few links eventually respect (unknown) communication delay bounds. Note that $\Omega$ also allows to solve consensus [23] and can in fact be implemented very efficiently. Another link-related assumption is used in the timeout-free implementation of $\mathcal{P}$ in [28], which requires a system where every correct processor is connected to a set of $f + 1$ processors via links that are not among their $f$ slowest ones. Finally, the perfect failure detector for the partially synchronous $\Theta$ model of [26] assumes that the maximum versus minimum end-to-end delays are unbounded but within some (known or unknown) $\Theta$ of each other.

The work most closely related to the present paper is the the adaptive failure detection protocol of [16], which uses a finite average delay assumption similar to (A2) to implement a failure detector that is perfect for a finite number of steps. It needs a clock and a finite average for arbitrary sequences of overlapping message round-trips in the entire system. This is a stronger property than we are assuming in this paper. In this paper we make weaker assumptions and provide a failure detector with stronger properties. Our paper also relates to the message classification model of [14], where slow vs. fast messages are distinguished in a time-free way.

6 Conclusion

We have shown that one can implement an eventually perfect failure detector in systems in which the average response time is finite and there exists at least one known operation (like incrementing an integer) that cannot be infinitely fast. Therefore, one can solve consensus deterministically in such systems. We described how to combine this new failure detector with other failure detectors to optimize the average behavior while still guaranteeing the properties of an eventually perfect failure detector.

References


A Appendix

This appendix provides some additional material to support reviewing. First, consensus protocols need variable-size stubborn channels instead of fixed-size stubborn channels. We show in Section A.1 how to implement a variable-size acknowledged stubborn channel on top of failure detector EA-FD (see Figure 4). Second, we show in Section A.2 that fixed-size stubborn channels can be implemented on top of reliable channels. This shows that fixed-size stubborn channels do not provide stronger properties than reliable channels.

A.1 Variable-Size Stubborn Channels

Typically, a round-based consensus protocol includes the round number within the messages it sends. Hence, correct protocols need to increase the size of messages if it takes a very large number of rounds to reach consensus. In this case, the size of a message $m$ is bounded by $O(\log(R))$, where $R$ is the number of rounds.

We show how to implement a stubborn channel for variable-size messages on top of the failure detector layer that we introduced in the last subsection and which itself runs on top of the stubborn channels for fixed-size messages that is provided by the FA model.

Messages of size $> S$ need to be fragmented. To do this, a sender $p_2$ splits a message in fragments of size $S - 1$ and adds a message tag of size 1. The message tag is 0 if more fragments will follow and 1 if it was the last fragment. As soon as the destination process has received all fragments (i.e., message tag is 1), the process delivers the message (see Figure 6). A message piggybacked on an acknowledgment message is also fragmented. The sender $p_2$ keeps sending poll messages (message tag 2) until it received the last fragment of the acknowledgment message (message flag 1). Process $p_2$ merges the fragments and delivers the acknowledgment message.
The pseudocode that shows how to implement the fragmentation and defragmentation is depicted in Figure 7. The property of the variables size channel is given by Definition 2 where $S = \infty$.

### A.2 Implementing Fixed-Size Reliable Channels

Fixed-size acknowledged stubborn channels can be implemented on top of reliable channels (see Figure 8). In this implementation we assume that a reliable channel provides (1) a primitive “$r_{send}(m)$ to $q$” to send a message $m$ to process $q$, and (2) a primitive “on $r_{deliver}(m)$ from $p$” to define an action to which messages are delivered. In the pseudocode we denote the request to execute a local action $A$ by the term “execute($A$)

```c
1 const int N; // number of processes
2 var
3 Msg SM[N] init undefined; /* next message to send */
4 Msg AM[N] init undefined; /* next msg to piggyback */
5 bool intransit [N] init false;

function sc_send (m) to p {
  if (! intransit [p]) {
    r_send ('msg', m) to p;
    intransit [p] = true;
  } else {
    SM[p] = m;
  }
}

function sc_piggyback(m,q) {
  AM[q] = m;
}

on r_deliver (n) from q {
  (tag, m) = n;
  if (tag == 'msg') {
    execute ('' on sc_deliver (m) from q ');
    r_send ('ack', AM[q]) to q;
    AM[q] = undefined;
  } else {
    execute ('' on sc_ack (m) from q ');
    if (SM[q] != undefined) {
      r_send ('msg', SM[q]) to q;
      SM[q] = undefined;
    } else {
      intransit [q] = false;
    }
  }
}
```

Figure 8: Implementation of fixed-size acknowledged stubborn channel on top of reliable channels.

Acknowledgments: We would like to thank Trevor Jim, Michael Merritt, and Francois Taiani for their very helpful discussions and suggestions.