

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

Christof FETZER
Computer Science Dept.
TU Dresden
D-01062 Dresden, Germany
christof.fetzer@inf.tu-dresden.de

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

Martin SÜSSKRAUT
Computer Science Dept.
TU Dresden
D-01062 Dresden, Germany
ms67@inf.tu-dresden.de

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 paper¹ 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. There 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, 9, 12, 11, 29]). The consensus problem is defined as follows:

¹A brief announcement appeared in the proceedings of PODC'04. The first author is supported by a Heinz-Nixdorf endowment. The work of the second author has been supported by the Austrian START-project Y41-MAT.

(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 Response time model or FAR 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 an unknown finite average response time and an unknown finite maximum speed for incrementing an integer. Despite of those weak assumptions, the FAR model permits a deterministic solution of the consensus problem. Note that there is a difference between a bounded average response time and a finite average response time. If we would assume a bounded average response time, there

would exist a finite constant B such that the average response time in all runs is at most B . However, by only assuming a finite average response time, we can find for any finite constant B a run in which the average response time is above B . Also note that a finite average response time does not bound individual response times - neither a priori nor a posteriori. In particular, we can find runs in which for each given constant D there exists a response time that is greater than D (example below).

The FAR model makes the following non-standard assumptions:

- *Incrementing an integer takes some unknown time $\geq G > 0$.*

In solving consensus, we will need to prevent the situation that a process is continuously getting faster over time. While it is difficult to enforce a minimum speed of a process, programs have naturally a finite speed because they are executed by a physical CPU. For example, the speed of incrementing an integer cannot become infinitely fast – even if one would constantly be upgrading the underlying hardware. Eventually, the speed of incrementing an integer will reach some physical or economic 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. For convenience, we chose that incrementing an integer has a finite speed. Note the existence of one operation with a finite speed would be sufficient.

- *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. While it is difficult to design programs that guarantee real-time deadlines, the response times of well architected distributed systems do not continuously increase. In a badly designed system, the response time of a process can go up continuously if other processes can flood this process with messages. In reality, most distributed applications are TCP-based and TCP provides some basic flow control. In the FAR model we make the following assumption:
- *The system enforces some basic flow control.* The system 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. Note in systems with reliable channels and without an application-level flow control mechanism, processes can exhibit longer and longer response times because “fast” processes can overload “slower” processes (see Section 6). Since our model has to be independent of the application behavior, we enforce some basic flow control.

The FLP model [17] 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 FAR model, we assume instead that the *average* response time is finite. To illustrate the difference, the FLP model permits that the transmission delay of any subsequence of the messages sent via a reliable channel to be monotonically increasing, e.g., 1, 2, 3, ... This is not allowed in the FAR model. Note however that such a behavior is allowed for infinite subsequences of messages sent via a link as long as there are sufficiently many faster messages sent via the same link 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 link 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.

We show in this paper that one can solve consensus deterministically in the FAR model. Instead of describing a consensus protocol, we show how to implement an eventually perfect failure detector [6]. Our failure detector can be used with an already published consensus protocol [20] 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 [6] is not a panacea for solving consensus. Assuming that the response times in large scale systems are eventually bounded is, for example, questionable

in the presence denial of service attacks. We believe that response times can increase proportionally to the duration of an attack. Even when using defense mechanisms in the Internet backbone, the duration of an attack can be arbitrarily prolonged by using a sufficiently large attack network. The advantage of the FAR model is that it permits to describe systems that can be attacked for arbitrary periods as long as the “gaps” between attacks periods are sufficiently large. Attackers need resources to mount an attack. Assuming that the attackers have finite resources, the gaps between increasing attacks need to increase too.

Based on the above discussion, we expect that the gaps between attacks will stay sufficiently large. Also in practice, in the “arms race” between the attackers and the defenders, the defenders appear to keep systems available “most of the time” even though some hosts might be unresponsive for a few hours or days during an attack. For example, even for high profile targets like the SCO website, the gaps between attacks are apparently sufficiently large. According to Netcraft, recent major attacks on SCO were in May 2003 (138 attack machines, 5:15 hours offline), in August 2003 (≥ 3 days offline), and December 2003 (≥ 8 hours offline) and February 2004 (≥ 400000 attack machines, 12 days offline). This means that even for such heavily attacked systems the FAR assumptions would be reasonable.

Also, very large systems (like P2P systems) might never stabilize completely. Ensuring and/or justifying “global” assumptions that response times are eventually bounded is in such systems intrinsically difficult. By contrast, in the FAR 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.

It might appear to some readers that the FAR model implies that a system will always stabilize for “sufficiently long periods, i.e., that the distance between two new maximum response times in the system would get further and further away from each other (because two successive maxima of a link need to be further and further away to keep the average response time finite and there are only a finite number of links). However, this is not true and one can construct runs in which the system never stabilizes. Consider a system consisting of two processes p and q connected by two links L (for requests from p to q and associated replies) and L' (for requests from q to p and associated replies). Say, p sent a request R to q via link L that has a longer response time than any previous request over L (i.e., a new maximum). During the period in which R is “in transit” the model permits q to exchange a sufficient number of requests via link L' to save up enough “credit” of a new maximum. As soon as p receives a reply/acknowledgment for R , there exists a request R' sent via L' which also will result in a new

maximum. In turn, while R' is in transit, a sufficient number of requests can be exchanged via L to save up credit for the next new maximum. Even though there is always a new maximum request in transit, the average response time of all links stay finite. In particular, the system will never stabilize – which is required in the partially synchronous models like [12, 6]. Note that in such a run, the eventually perfect failure detectors that have been proposed for partially synchronous models will never become perfect!

The advantage of our failure detector is that given the assumptions of the FAR model, one can prove that algorithms like the consensus algorithm of [20] are correct and in particular, will always terminate. The price to pay for correctness are potentially very long detection delays because correct processes can be very slow. However, 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 (for “single-shot” algorithms [18] like consensus) without sacrificing the guaranteed termination.

Outline: We introduce the FAR 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 describes our performance measurements. Section 7 concludes the paper.

2 Finite Average Response Time Model

This section presents our new asynchronous system model (FAR 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 [20]. Note that stubborn channels do not strengthen our model in comparison to the FLP model². The main advantage of stubborn channels over reliable channels is that they enforce some basic flow control and can be implemented atop of unreliable channels with a bounded amount of memory and a bounded number of messages in transit at any time.

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

² We show in [16] that stubborn channels can be implemented using reliable channels.

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 $correct_p$) are defined with respect to an implicitly given run.

Definition 1 (correct). We define that predicate $correct_p$ 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 $\leq S$ to a correct process q via an acknowledged stubborn channel and p delays sending any other message to q until it receives an acknowledgment for m , then eventually m will be delivered to q and p will receive an acknowledgment that q has delivered m .

Definition 3 (fixed-size). For $S < \infty$, 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 $SC_{p \rightarrow q}$.

Operational Aspects. Instead of 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 sequence³. The delivery of messages arriving while another messages is processed is deferred to sequentialize the delivery of messages. Operationally, we define the following interface for fixed-size stubborn channels (see Figure 1):

- To send a message of size $\leq S$, a process can call a primitive sc_send . By expression “ $sc_send(m)$ 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., $size(m) \leq S$.
- A message m from p to q is delivered by an action on q . In the pseudocode we express this delivery action as follows: “ $on\ sc_deliver\ m\ from\ p\ \{ \dots\ sc_piggyback(n)\ to\ p;\ \dots\}$ ”. After executing this action, q sends an acknowledgment to p . 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., $size(n) \leq S$.

³To simplify the pseudo-code, an action can execute another action in the same way an action would call a function. An action and also nested actions 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: $on\ sc_ack(n)\ from\ q\ \{ \dots\}$. Note that the message n that was piggybacked on the acknowledgment message is not acknowledged.

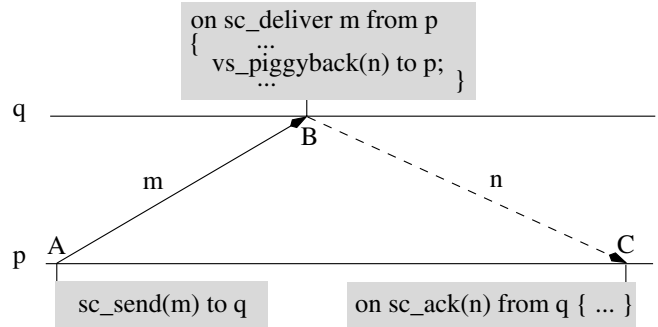


Figure 1: Communication is push based: The arrival of a messages automatically triggers the execution of an action. The reception and processing of a message is acknowledged. We permit application messages to be piggybacked on an acknowledgment message. Note however that these piggybacked messages are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

Response Time. The FAR 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 it is the real-time duration between point A and C.

To make our assumption as weak as possible, the FAR 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.

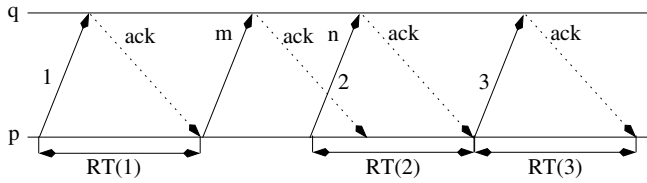


Figure 2: The response time of a message is the real-time duration between the time at which a message is sent until the time the acknowledgment is delivered. Unacknowledged messages are not enumerated.

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 FAR 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 two fixed-size acknowledged stubborn channel $C_{p \rightarrow q}$ and $C_{q \rightarrow p}$ of size S .
- (A2) The average response time converges⁴ and it is finite:

$$\forall p \forall q : \lim_{i \rightarrow |I_{p \rightarrow q}|} \frac{\sum_{1 \leq j \leq i} RT_{p \rightarrow q}(j)}{i} < \infty$$

- (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 FAR 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.

⁴ Using weaker forms of convergence than that defined by the limit are possible but for simplicity we use the standard definition.

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 (e.g., a sleep) for which there exists an unknown but positive minimum execution time would be sufficient. Note that the minimum response time of a message is permitted to be smaller than G (by avoiding incrementing integers by 1, e.g., by using subtractions by -1 if needed).

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, $I_{p \rightarrow q}$ only contains messages that are acknowledged. Hence, if the number of messages sent via a channel $SC_{p \rightarrow q}$ is finite (i.e., $|I_{p \rightarrow q}| < \infty$), the average response time is always finite. The FAR 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.

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 FAR 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 FAR model that is still sufficiently strong to implement the failure detector. In addition, consensus protocols need to send variable size messages. (In [16] we show how to implement variable-size stubborn channels on top of the failure detector layer.)

3.1 Weak Clock

The FAR 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 semantics (see Figure 3):

$$(W) \forall p, \forall s, \forall t : s < t \wedge \text{correct}_p \Rightarrow$$

$$0 \leq W_p^t - W_p^s \leq \lceil \frac{t-s}{G} \rceil$$

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 proceeds⁵.

A correct process will execute action *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.

```

1 var int c = 1;
2 function W() { return c; }
3 function tick () { c++; }
4 Action Tick { tick (); }

```

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 *tick* takes at least some unknown time $G > 0$ because the function increments variable c (line 3). Variable c is only modified by function *tick* and assumption (A0) implies that action *Tick* and hence function *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 *Tick* infinitely often, for each B we can find a time t such that p has executed *Tick* at least B times by time t . In other words, property (M) holds.

3.2 Eventually Perfect Failure Detector EA-FD

Our eventually perfect failure detector *EA-FD* (see Figure 4) is based on the following idea: *EA-FD* measures the response time of messages sent by the application using the

⁵Typically, one would however use the real-time clock of a computer. The nice property of the FAR model is that it copes with fast or slow clocks.

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 $is_suspected(q)$ 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 $is_suspected(q)$, 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 EA-FD is an eventually perfect failure detector.*

Proof. To show that EA-FD is complete, we have to show the following: If a correct process q calls $is_suspected(p)$ an unbounded number of times for a crashed process p , eventually $is_suspected(p)$ 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, \dots of actions which (1) all happened after the crash of p , (2) all acknowledgments from p to q have

```

1 const int N; /* # procs */
2 var int unacked[N] init 0; /* send-time of last unacked msg */
3 Msg fdack[N] init undefined; /* piggyback msg */
4 Msg nextmsg[N] init undefined; /* next msg to send */
5 int slowmsgs[N] = 0; /* total number of slow msgs */
6 int numFast[N] = 0; /* # fast msgs btwn two slow msgs */
7
8 function fd_timeout(process q) {
9   return (1+numFast[q])*(1+log(1+slowmsgs[q]));
10 }
11 function is_suspected(q) {
12   send_next(q);
13   if (!unacked[q] or q == myself)
14     return false;
15   else
16     return unacked[q]+fd_timeout(q) < W();
17 }
18 function send_next(q) {
19   if (!unacked[q]) {
20     unacked[q] = W();
21     sc_send(nextmsg[q]) to q;
22     nextmsg[q] = undefined;
23   }
24 }
25 function fd_send(n) to q {
26   nextmsg[q] = n;
27   send_next(q);
28 }
29 function fd_piggyback(n) to p {
30   fdack[p] = n;
31 }
32 on sc_ack(m) from q {
33   if (is_suspected(q)) {
34     slowmsgs[q]++;
35     numFast[q] = 0;
36   } else {
37     numFast[q]++;
38   }
39   unacked[q] = 0;
40   execute(fd_ack(m) from q);
41   send_next(q);
42 }
43 on sc_deliver(m) from p {
44   if (m != undefined) {
45     execute(fd_deliver(m) from p);
46   }
47   sc_piggyback(fdack[p]) to p;
48   fdack[p] = undefined;
49 }

```

Figure 4: Code of eventually perfect failure detector EA-FD. This code is a layer between the application and the fixed-size stubborn channels provided by the FAR model. The layer running on this code must use primitives *fd_send*, *fd_piggyback*, etc instead of primitives *sc_send*, *sc_piggyback*, etc.

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 unacknowledged 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

$$\text{unacked}[p] + (1 + \text{numFast}[p]) * (1 + \log(1 + \text{slowmsgs}[p])) < W()$$

which means that in all actions $A_{j,j \geq i}$ process p is suspected by q .

To show that EA-FD 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 suspects a correct process q infinitely often. A wrong suspicion 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 selection 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 acknowledgment of the k -th slow message is therefore $\sum_{0 \leq j \leq k} (1 + \text{numFast}_p^j)$. The lower bound for the response time of the j -th slow message is $G(1 + \text{numFast}_p^j)(1 + \log(j+1))$. Hence, we know that:

$$\lim_{i \rightarrow |I_{q \rightarrow p}|} \frac{\sum_{1 \leq j \leq i} RT_{q \rightarrow p}(j)}{i} \geq \lim_{k \rightarrow \infty} \frac{G \sum_{0 \leq j \leq k} (1 + \text{numFast}_p^j)(1 + \log(j+1))}{\sum_{0 \leq j \leq k} (1 + \text{numFast}_p^j)} = \infty.$$

This is a contradiction to assumption (A2). \square

4 Practical Aspects

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., [15, 3, 7]) but which does not guarantee termination in the FAR 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).

```

1 const int FUSION_THRESHOLD;
2
3 function fd_timeout (process p) {
4     if (slowmsgs[p] < FUSION_THRESHOLD) {
5         return qos_fd_timeout ();
6     } else {
7         return (1+numFast[p])*(1+log(1+slowmsgs[q]));
8     }
9 }

```

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.

5 Related Work

The impossibility of deterministic consensus in the FLP model [17] stimulated a wealth of research. In [9], 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 [12] 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 [6].

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 [5], which add an oracle that provides processes with hints about crashed processes. [4] provides the weakest failure detector for solving consensus (when a majority of processes stay correct), and [6] 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 [6, 23, 19, 24, 25, 10, 30, 26, 7, 1, 21, 15, 22, 3, 2, 28] rely upon the generalized GST model of [6], or even a synchronous model. The simple implementation of an eventually perfect failure detector $\diamond\mathcal{P}$ in [6] is based upon mon-

itoring 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 $\diamond\mathcal{S}$ was derived in [26]. 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 [25] 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 [14], which allow to solve consensus in the timed asynchronous model [8], and the fast failure detectors of [22], 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. [26, 15], 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 Ω 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 Ω also allows to solve consensus [24] 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 Θ model of [27] assumes that the maximum versus minimum end-to-end delays are unbounded but within some (known or unknown) Θ of each other.

The work most closely related to the present paper is the adaptive failure detection protocol of [15], 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 [13], where slow vs. fast messages are distinguished in a time-free way.

6 Performance

We performed two experiments to evaluate the assumptions of the FAR model. First, we measured the average response time for systems under attack. Second, we measured the average response time of a client/server system connected by a reliable channel (and without an application level flow control mechanism) and of a client/server system connected by a stubborn channel. Our measurement programs were implemented in Java and we executed the measurements on a set of computers (Athlon 64 3200+, 1 GByte RAM each) connected by a 1 Gbit Ethernet switch. In both our experiments, clients generated requests (a small random integer) and a server replied to these requests (with a prime factorization of the integer). The processing time of such a request takes in average less than 10 ms (see Figure 7).

We simulated a denial of service attack by two clients sending requests to a server as fast as possible, i.e., without flow control. We measured the response time with the help of a third client (with flow control). The attack of the two clients is coordinated in the sense that they synchronize the periods in which they attack. The attack duration and the gap between two consecutive attacks is doubled after each attack. Figure 6 shows that the average response time peaks after a few attacks.

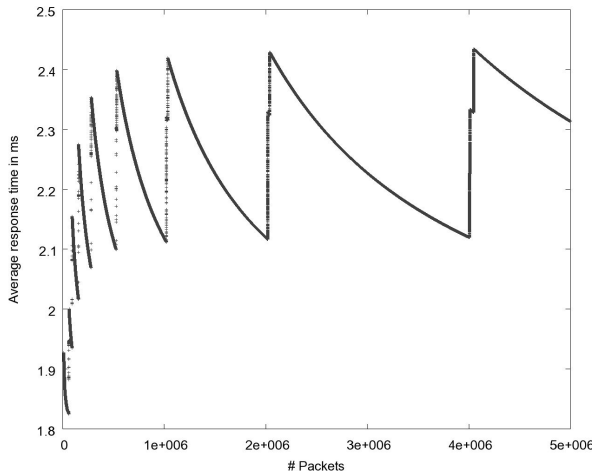


Figure 6: Response time average of a link to a host that is under attack by two machines (8 attack periods). Both the duration of attacks and the gap between attacks are doubling after each attack.

Our choice of using stubborn channels instead of reliable channels in the FAR model was based on the observation that reliable channels do not necessarily bound the average response time. We illustrate this observation with the following experiment. A client sent 100000 requests to a server. Figure 7 shows that the client was able to flood the server with requests when using a reliable channel: the aver-

age response time via the reliable channel was continuously increasing. However, the response time of the stubborn channel was slightly decreasing. The decrease can be explained by caching effects that help to reduce the response time.

The total runtime of the experiment using reliable channels was 498.75 seconds. Note that even though a stubborn channel acknowledges every message, the total runtime of this experiment for stubborn channels was only 168.45 seconds.

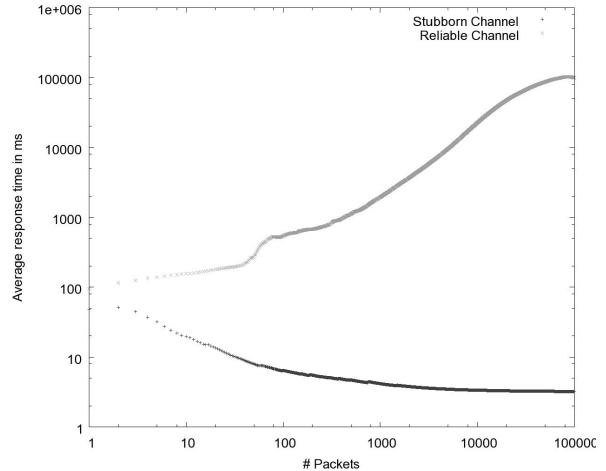


Figure 7: The average response time of two processes connected by a reliable channel without flow control can go up continuously (upper curve). The flow control of the stubborn channel kept the average response time finite (and in this case, caching reduced the response time slightly ; lower curve).

7 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

- [1] M. K. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal of Computing*, 29(6):2040–2073, April 2000.
- [2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, 2003. (to appear).

- [3] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363. IEEE Computer Society, 2002.
- [4] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 147–158, Aug 1992.
- [5] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug 1991.
- [6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [7] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, 2002.
- [8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, Jun 1999.
- [9] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987.
- [10] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings 3rd European Dependable Computing Conference (EDCC-3)*, volume 1667 of LNCS 1667, pages 71–87. Springer, September 1999.
- [11] P. Dutta and R. Guerraoui. The inherent price of indulgence. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, pages 88–97. ACM Press, 2002.
- [12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.
- [13] C. Fetzer. The message classification model. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 153–162. ACM Press, 1998.
- [14] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions of Computers*, 52:99–112, Feb 2003.
- [15] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, Seoul, Korea, Dec 2001.
- [16] C. Fetzer and U. Schmid. On the possibility of consensus in asynchronous systems with finite average response times. Technical Report 14/2004, 2004.
- [17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, Apr. 1985.
- [18] R. Friedman, A. Mostefaoui, and M. Raynal. The notion of veto number and the respective power of $\diamond p$ and $\diamond s$ to solve one-shot agreement problems. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (DISC04)*, pages 41–54. Springer Verlag, October 2004.
- [19] V. K. Garg and J. R. Mitchell. Implementable failure detectors in asynchronous systems. In *Proceedings of the 18th Int. Conference on Foundations of Software Technology and Theoretical Computer Science (FST & TCS'98)*, LNCS 1530, pages 158–169. Springer, 1998.
- [20] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels, 1996.
- [21] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, pages 170–179, Aug. 2001.
- [22] J.-F. Hermant and G. Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, Aug. 2002.
- [23] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, Dec. 1997.
- [24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [25] M. Larrea, A. Fernández, and S. Arévalo. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, LNCS 1693, pages 34–48. Springer, Sept. 1999.
- [26] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, page 334, 2000.
- [27] G. Le Lann and U. Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, January 2003.
- [28] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, San Francisco, CA, June 22–25, 2003.
- [29] A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, 2003.
- [30] A. Mostéfaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings 13th International Symposium on Distributed Computing (DISC'99)*, volume 1693 of LNCS, pages 49–63. Springer-Verlag, 1999.