

Perfect Failure Detection In Timed Asynchronous Systems

Christof Fetzer
 AT&T Labs Research
 180 Park Ave
 Florham-Park, NJ 07930, USA
 christof@research.att.com

Abstract— Perfect failure detectors can correctly decide whether a computer is crashed. However, it is impossible to implement a perfect failure detector in purely asynchronous systems. We show how to enforce perfect failure detection in timed asynchronous systems with hardware watchdogs. The two main system model assumptions are (1) each computer can measure time intervals with a known maximum error, and (2) each computer has a watchdog that crashes the computer unless the watchdog is periodically updated. We have implemented a system that satisfies both assumptions using a combination of off-the-shelf software and hardware. To implement a perfect failure detector for process crash failures, we show that in some systems a hardware watchdog is actually not necessary.

Keywords: perfect failure detection, crash failures, asynchronous distributed systems, timed asynchronous system model.

I. INTRODUCTION

A *perfect failure detector* [5] can *correctly* decide whether a computer is crashed. Perfect failure detectors can be very useful in the design of fault-tolerant distributed systems. For example, a backup system can use a perfect failure detector to detect crash failures of the primary system. In case the backup detects a crash failure, it can take over without risking that the primary is still alive – which could lead to undesirable inconsistencies.

Most existing failure detectors are heartbeat based and do not prevent wrong suspicions. A wrong suspicion might lead to situations in which two computers think that they are the primary computer. Some membership based systems like ISIS [2] detect wrong suspicions and they remove wrongly suspected processes from the membership. However, these processes might learn of the removal after they are suspected. We will explain why this is not sufficient for some systems, e.g., systems that access external devices like shared disks.

In this paper we propose a novel protocol that makes sure that computers are crashed before they are suspected. The difficulty in achieving this is that if a computer is suspected, it is crashed or partitioned away from the other computers, e.g., due to a network failure. In the latter case, communication with the computer is lost and hence, initiating a remote crash is not possible. More formally, we will explain why this problem is not solvable in purely asynchronous systems like those

To appear in IEEE Transactions on Computers. An earlier version of this paper appeared in the 21st Proceedings of the International Conference on Distributed Computing Systems (ICDCS2001), April 2001, Phoenix, AZ, pp. 350-357.

described by the FLP model [12].

Our failure detector protocol is a two level protocol. The lower-level level implements an unreliable failure detector that is supposed to minimize wrong suspicions, i.e., it should only rarely suspect non-crashed computers. However, one cannot completely prevent that wrong suspicions happen on this level. The higher-level mechanism makes sure that before any suspicion is propagated to the clients of the failure detector, the suspected computer is guaranteed to have failed using leases and the watchdog of the suspected computer.

Originally, perfect failure detection was defined for detecting *process* crash failures in systems in which processes *cannot* recover from a crash. In this paper we are interested in detecting *computer* crash failures in systems in which computers *can* recover from a crash failure. We are also interested in detecting crash failures within a bounded time. To cope with these constraints, we define a new failure detector class \mathcal{TP} (where \mathcal{T} stands for timed and \mathcal{P} for perfect). A failure detector class is the set of all failure detectors that satisfy a given set of properties. We call a failure detector in \mathcal{TP} *timed-perfect*. Each timed-perfect failure detector correctly decides if a computer c is crashed or not – even if somebody permanently or transiently disconnects the network of c . Formally, we show in this paper that each timed-perfect failure detector satisfies the conditions of a perfect failure detector in systems without recovery.

The remaining sections of this paper are organized as follows. We sketch in Section II how one can implement a perfect failure detector in a synchronous system before we explain why one cannot implement a perfect failure detector in purely asynchronous systems. In Section III we describe three sample application domains of perfect failure detectors to support that they are useful in the design of fault-tolerant distributed systems. We define in Section IV timed-perfect failure detectors and show in Section V that timed-perfect failure detectors are indeed perfect in systems without recoveries. We introduce the timed asynchronous system model in Section VI before we describe the implementation of a timed-perfect failure detector in Section VII. We describe performance measurements of our protocol in Section VIII and explain why one does not necessarily need a hardware watchdog for a perfect process failure detector in Section IX. Before we conclude the paper, we review related work in Section X.

```

1  function SP(computer c)
   send {ping} to c;
   wait
5   on receive {pong} from c
   return up;
   after  $2^*\delta$ 
   return crashed;

10 body
   forever
   on receive {ping} from sender
   send {pong} to sender;

```

Fig. 1. A perfect failure detector protocol SP for completely synchronous systems. To query the status of a computer e , a process calls function $SP(e)$.

II. POSSIBILITY AND IMPOSSIBILITY RESULTS

In this section we first explain how one can implement a perfect failure detector in a completely synchronous system before we show why it cannot be implemented in a purely asynchronous system.

A. Completely Synchronous Systems

A protocol that implements a perfect failure detector in completely synchronous distributed systems is depicted in Figure 1. A completely synchronous system is a distributed system in which the maximum communication and processing delays are known and the only failures permitted are crash failures. For simplicity, in this section we assume that crashed computers do not recover, i.e., they stay crashed forever. In such systems there exists an a priori known time constant δ such that each message sent by a non-crashed computer d to another non-crashed computer c is received and processed by c within δ time units. Therefore, c will reply to a message from d and d will receive and process c 's reply within 2δ time units.

A failure detector FD consists of a set of failure detector modules: each computer d is running a failure detector module FD_d . Processes running on computer d can call module FD_d to see if FD_d suspects some other computer c to be crashed. A *perfect failure detector* never suspects a non-crashed computer and will eventually suspect all crashed computers. In what follows, c denotes a computer, d denotes a computer that wants to detect when c crashes, and e denotes a third computer.

The failure detector implemented by the protocol depicted in Figure 1 is called SP , where S stands for synchronous and P for perfect. The protocol works as follows. Upon being queried for the status of a computer c , the failure detector module SP_d sends a ‘‘ping’’ message to c . Module SP_d waits for at least 2δ time units for a reply from c (line 6 of Figure 1). If SP_d receives a reply within 2δ , SP_d returns a constant **up** to indicate that c is not crashed. Otherwise, SP_d correctly suspects c to have crashed by returning a constant **crashed**. In completely synchronous systems this protocol suspects all crashed computers and it does not suspect non-crashed computers. Therefore, this protocol implements a perfect failure detector in completely synchronous systems.

B. Asynchronous Systems

Most of the existing distributed systems are not completely synchronous. In particular, no matter how large δ is chosen, as long as δ is finite there is no guarantee that a non-crashed computer d receives a reply from another non-crashed c within 2δ time units. For example, consider the situation in which somebody accidentally disconnects the network cables of c for more than 2δ time units. After 2δ time units SP_d will incorrectly suspect c . Thus, SP_d might violate the properties of a perfect failure detector in non-synchronous systems.

More precisely, in purely asynchronous systems one can show that it is impossible to implement a perfect failure detector. Purely asynchronous systems are characterized by having no finite upper bound on the transmission delay of messages nor having a finite upper bound on the processing delays. Also, these systems do not have access to a clock that permits them to measure real-time.

Any perfect failure detector is *complete* and *accurate*. A *complete* failure detector will eventually detect all crashed computers and an *accurate* failure detector will at no point suspect a computer that is not (yet) crashed. To explain that one cannot implement a perfect failure detector in a purely asynchronous system, let us assume there exists such a failure detector AP . We use AP to derive a contradiction such that whatever AP returns as the status of a process c (i.e., returning either crashed or up) at a carefully selected time t , it will be wrong.

We select an execution E of the protocol such that a computer c crashes at some time s and a computer d never crashes in E . Since AP is perfect, it is by definition complete and hence, AP_d will eventually suspect c at some time $t \geq s$. Now consider a run E' which is the same as E up to time t but that c instead of crashing at time s , it just does not perform any step in $[s, t]$. Since E' is indistinguishable from E until t , AP_d will also suspect c at t in E' . This is a contradiction to our assumption that AP is perfect.

This shows informally that one cannot implement a perfect failure detector in a purely asynchronous system. Before we show how to circumvent this impossibility result in timed asynchronous systems, we first explain why perfect failure detection is of practical relevance.

III. EXAMPLE APPLICATIONS

To explain why perfect failure detectors are useful in the design of fault-tolerant distributed applications, we first sketch a practical problem that can be solved with a perfect failure detector but not with a failure detector that only ‘‘simulates’’ the behavior of a perfect failure detector. We conclude this section by sketching two more application domains of perfect failure detectors.

A. I/O Fencing

Consider that computers c and d share a disk via an external bus, e.g., both computers can access the disk via a shared SCSI bus [1]. As long as c is not crashed, it has exclusive ac-

cess to the disk. If c crashes, d , which is the backup of c , will need to access the disk. If d would wrongly suspect c , d and c might access the disk in parallel which could lead to inconsistent data. Note that the communication with the disk is not message based. Typically, the access to the disk is via memory mapped I/O, e.g., a computer can communicate with the disk by writing and reading some memory mapped registers of the SCSI controller.

The above described problem of preventing a wrongly suspected computer from accessing a shared hardware device is known as I/O fencing and is addressed by modern distributed file systems like the *Global File System* [16]. There exists several solutions that address this problem. We classify these as follows:

- **Reservation Based:** The shared hardware device provides functionality that permits computer to prevent access by the other computer. Some SCSI devices and Fibre Channel switches support this functionality [16].
- **SONITH based:** A SONITH (Shoot The Other Node In The Head) approach requires a network power switch (e.g., a X10 based switch) and before a computer d accesses the shared disk, it communicates with the power switch to shut off the suspected computer c .

Our proposed perfect failure detector is similar to that of the SONITH based solutions in that a wrongly suspected computer is crashed to guarantee consistency. Note that SONITH based approaches need reliable communication to the network power switch to be able to shut off the suspected computer.

We will show that our protocol does not need reliable communication to enforce perfect failure detection since it is a lease based approach. This can make our system much more robust than for example X10 based systems are that are known for their low reliability due to the noise on the power lines, interference, and missing feedback from the X10 switches (1-way switches). Also, sometimes communication is only possible through the suspected device, e.g., if this device routes messages (see [10] for details). Moreover, we show in Sections VI and IX that our protocol does not always need special hardware, i.e., does not always need a hardware watchdog.

B. Simulated Perfect Failure Detectors

A simulated perfect failure detector is not sufficient to solve the I/O fencing problem. Simulating a perfect failure detector means that internal observers (i.e., all computers except c) cannot detect when a computer c is wrongly suspected [19]. Intuitively, a non-crashed computer c will eventually learn that it is wrongly suspected and will therefore crash itself to make sure that other processes cannot detect that the failure detector made a mistake. However, it is not guaranteed that c learns of such a suspicion before the other computers actually suspect d . To explain this, consider that we can find an execution similar to that of E' in Section II-B. In such an execution, c must be suspected at some time t to ensure completeness. However, c cannot have learned of this suspicion before t because we can select E' such that c is not processing any events until after t .

Most group membership protocols provide guarantees similar to a simulated perfect failure detector since suspected processes are excluded from the membership and these processes will eventually learn that they are suspected. However, the guarantees provided by a simulated perfect failure detector is not sufficient to solve the above described I/O fencing problem.

Consider that c is suspected at time t by computer d but c does not learn until time $u > t$ that d suspects c . Hence, during the time interval $[t, u)$ both computers might access the external shared disk since c does not know that is suspected and d takes over since c has supposedly crashed. Of course, the goal of using a failure detector in this application is to avoid just this situation!

A “real” perfect failure detector can solve this problem since when d 's failure detector says that c is suspected at t , then c is indeed crashed at t . A simulated perfect failure detector is often not sufficient if the system contains external observers (like the disk) or hidden communication channels. The latter is related to the problem of tracking causality in systems with hidden communication channels as discussed in [7].

C. Other Application Domains

Perfect failure detectors can be used to solve the I/O fencing problem. In addition to synchronizing the access to shared physical resources, they can also be used in synchronizing the access to shared logical resources. We sketch below how one can use a perfect failure detector in the implementation of an IP address take over mechanism. In this example, an IP address is a shared logical resource. The perfect failure detector is used to enforce mutual exclusion such that at most one computer is using the IP address at a time. The last application domain that we describe is on-demand rejuvenation of computers. This domain is specific to our protocol in the sense that the protocol enforces perfect failure detection by rejuvenating unresponsive computers.

IP Address Take Over

An *IP address take-over* mechanism enforces that a computer d takes over the IP address of a computer c after c has failed. Such a mechanism facilitates the construction of simple but practical fault-tolerant systems. For example, this can be used to make cluster systems more fault-tolerant. Clusters often have one “head-end” computer, say, c that is responsible for the communication with external systems. Users might have to log in to this computer to submit jobs to the cluster. Communication with computer c is typically TCP/IP or UDP/IP based and external machines know the IP address IP_c of computer c . To avoid that communication with the cluster becomes unavailable in case c goes down, clusters might have a second computer d that takes over as a head-end if c crashes.

Keeping the IP address of the head-end computer constant simplifies the cluster design. Otherwise, external and internal machines would have to update the IP address IP_c . This is difficult since the address is typically cached in multiple ser-

vices, e.g., the DNS service and client programs. It is therefore easier to assign address IP_c to d instead. However, one wants to avoid that c and d use IP_c at the same point in time. Otherwise, various inconsistencies could occur. For example, a constant loss of all client connections due to multiple inconsistent replies by c and d , garbled reply messages by fusing network packets from c and d , or possibly parallel execution of commands with at most once semantics by c and d .

The use of a perfect failure detector permits one to ensure that at any point in time at most one of the computers uses IP_c . Our implementation of an IP address take-over mechanism addresses a range of technical difficulties that are beyond the scope of this paper. For example, we keep the mapping between IP_c and the underlying Ethernet address constant since changing this mapping can lead to various problems as already pointed out in [1]. To do this, we use the feature that the software can change the Ethernet address of network interface cards.

On Demand Rejuvenation

As [15] points out, the state of computers typically ages and rejuvenating the state of a computer periodically (e.g., by rebooting it) can actually increase the availability of the computer. Instead of performing periodic reboots, one can use the protocol proposed in this paper to perform an on-demand rejuvenation. Failures of a computer are detected and failures due to state aging are automatically “repaired” by rebooting the computer. Such a system (but with a different protocol) is described in [10].

IV. SPECIFICATION OF \mathcal{TP}

Originally, perfect failure detectors [5] were defined to augment purely asynchronous systems [12]. The model of [5] does not consider crash recovery. In the systems we are interested in, computers do typically recover after a crash. To solve this problem, one could of course assign each computer a new identity after it recovers. Often, this might be an acceptable solution. We show in Section V that our failure detector satisfies the specification of [5].

However, in practice one sometimes would like to keep the identities of computers constant. This permits clients to cache the identities of computers without the need for a protocol that keeps the cache consistent. For example, some computers cache the IP addresses of the local DNS servers. These servers are identified by their IP addresses and they keep these IP addresses constant even after they reboot. To address computer recoveries without changing the identity of recovering computers, we define a new class of failure detectors.

Often, system designers are interested in specifying an upper bound on the maximum detection delay of a crash failure. This might be needed to maximize the system availability, or to enforce maximum response times that are imposed by the system requirements. To satisfy this need, our specification includes also an upper bound on the detection delay instead of requiring that crashes are detected eventually.

We define a new timed-perfect failure detector class \mathcal{TP} .

The failure detectors in \mathcal{TP} detect crash failures and recoveries within a known maximum time. The definition of the timed-perfect failure detector class is quite different from the original definition of the perfect failure detector class [5]. Nevertheless, in systems in which computers cannot recover from a crash, each failure detector in \mathcal{TP} satisfies all properties of a perfect failure detector (see Section V).

Consider a system consisting of at least three computers c , d , and e . The number of computers is finite. We use real-time values (denoted by s , t , u , and v) in the specification of \mathcal{TP} . A computer is either *crashed* or *up*. We assume that computers can recover from a crash. For simplicity, we assume that time starts with 0 and that we can subtract any two time values: if $u < v$, then $u - v$ is defined to be 0. Intuitively, the system starts at time 0 – but there is no requirement that any computer is actually up at time 0.

Let TP be a failure detector in \mathcal{TP} . At any time t , computer d ’s failure detector module TP_d defines for each computer c an output value $TP_d^t(c)$. This output value of TP_d is either **crashed**, **up**, or **recovering**. Intuitively, TP_d outputs **crashed** if TP_d suspects c to be crashed, TP_d outputs **up** if TP_d does not suspect c to be crashed, and TP_d outputs **recovering** if d has just recovered from a crash and TP_d is not yet fully initialized.

The failure detector class \mathcal{TP} is defined by three properties and the usage of a finite constant $DD \geq 0$ (Detection Delay).

Property Crash Accuracy: For any time u such that $TP_d^u(c) = \mathbf{crashed}$, there exists a $t \in [u - DD, u]$ such that c is crashed at t .

Property Up Accuracy: For any time u such that $TP_d^u(c) = \mathbf{up}$, there exists a $t \in [u - DD, u]$ such that c is up at t .

Property Recovering Accuracy: For any time u such that $TP_d^u(c) = \mathbf{recovering}$, there exists a $t \in [u - DD, u]$ such that either d is crashed at t or $u - DD = 0$, i.e., the system has just started.

Note that we replaced the traditional strong completeness property (“eventually each crashed computer is permanently suspected”) by two accuracy properties: **Up Accuracy** and **Recovering Accuracy**. These two accuracy properties state that a computer c that has been crashed for more than DD time units can neither be classified as **up** nor as **recovering**. Therefore, c ’s status has to be classified as **crashed** as long as c stays crashed.

If the status of a computer c changes (e.g., it recovers from a crash), the failure detector has DD time units to correct the classification of c . Note however that as soon as a computer c stays permanently in a state (i.e., up or crashed), a timed-perfect failure detector has to classify c within DD time units correctly and permanently as either **up** or **crashed**.

V. PERFECT FAILURE DETECTORS

The definition of the timed-perfect failure detector class is quite different from the original definition of the perfect failure detector class [5]. We show that timed-perfect failure de-

tectors are still perfect with respect to the original definition: if no computer recovers from a crash, a timed-perfect failure detector satisfies the original properties of a perfect failure detector.

A. Definition of \mathcal{P}

The class of perfect failure detectors \mathcal{P} was originally specified by two properties [5]: **Strong Completeness** and **Strong Accuracy**. In the notation used in this paper, one can specify these properties as follows. Let P be a failure detector in \mathcal{P} , let $CRASHED$ be the set of all computers that ever crash, and UP the set of all computers that never crash.

Strong Completeness: *Eventually each computer that is crashed is permanently suspected by all non-crashed computers.* $\exists t, \forall c \in CRASHED, \forall d \in UP, \forall u \geq t: P_d^u(c) = \text{crashed}$.

Strong Accuracy: *No computer is suspected before it crashes.* $\forall t, \forall c, \forall d$: if c and d are up at t , then $P_d^t(c) \neq \text{crashed}$.

An underlying assumption of this original specification is that computers do not recover:

No Recovery: *For all times t , if a computer c is crashed at time t , then c is crashed at all times $u \geq t$.*

B. Relationship between \mathcal{P} and \mathcal{TP}

For system in which computers cannot recover from a crash, a timed-perfect failure detector satisfies the **Strong Completeness** and the **Strong Accuracy** properties. In other words, each failure detector in class \mathcal{TP} is a perfect failure detector.

Theorem: *In systems without recoveries, each timed-perfect failure detector is perfect.*

Proof: We have to show that if the **No Recovery** property holds, then each timed-perfect failure detector satisfies properties **Strong Completeness** and **Strong Accuracy**. Let TP a timed-perfect failure detector. Since TP is timed-perfect, it satisfies the properties **Crash Accuracy**, **Up Accuracy**, and **Recovering Accuracy**.

Let d be a computer that starts at time 0 and that never crashes, i.e., $d \in UP$, and let c be a computer that crashes at some time $s \geq 0$, i.e., $c \in CRASHED$. The **No Recovery** property implies that c is crashed at all times $v \geq s$. Hence, for all times $v > s + DD$, property **Up Accuracy** implies that $TP_d^v(c) \neq \text{up}$ since c has been crashed for more than DD time units. Since $d \in UP$ has been up for more than DD time units at all times $v > DD$, property **Recovering Accuracy** states that $TP_d^v(c) \neq \text{recovering}$. Since the output of TP is either **crashed**, **up**, or **recovering**, for all times $v > s + DD$, $TP_d^v(c) = \text{crashed}$.

Since the number of computers is finite, there exists a time u such that all computers that will eventually crash are already crashed at time u . After time $t := u + DD$ all non-crashed computers suspect all crashed computers. Therefore, property **Strong Completeness** holds.

Let d be a computer that suspects c at time v , i.e., $TP_d^v(c) = \text{crashed}$. Property **Crash Accuracy** implies that

there exists a $s \leq v$ such that c is crashed at s . Property **No Recovery** states that c stays crashed at all times $u \geq s$. Hence, c is crashed at time v . Therefore, property **Strong Accuracy** holds. \square

VI. TIMED SYSTEMS

In timed asynchronous distributed systems [8] (short: timed systems) there exists no upper bound on the transmission delay of messages nor on the execution times of protocols. Also, computers can become transiently or permanently partitioned. Each computer has a local (unsynchronized) hardware clock. A hardware clock allows the measurement of time intervals with a known maximum error. In this paper we extend the timed model by hardware watchdogs that permit a non-responsive computer to be reset as soon as the value of the hardware clock reaches a certain value. In timed systems with hardware watchdogs there is exactly one timeliness guarantee: a computer can be crashed by a given time. Hardware clocks and watchdogs are available as off-the-shelf boards but also as software modules. For example, the Linux kernel 2.4.x includes several watchdog drivers that include a software watchdog and drivers for several hardware watchdog boards.

More precisely, a timed system consists of a finite set of computers. Each computer e has access to a local hardware clock H_e with a bounded drift rate. The value of H_e at real-time t is denoted by H_e^t . The drift rate of a hardware clock is bounded by a known constant ρ :

$$\forall s, \forall t \geq s : (t - s)(1 - \rho) \leq H_e^t - H_e^s \leq (t - s)(1 + \rho)$$

We use the term “local time” T of a computer e to denote a point in real-time t such that $H_e^t = T$. Local time values of a computer are denoted by T , U , and V . We use the term “ticks” to refer to the duration of local time intervals: the duration of local time interval $[T, U]$ is $U - T$ ticks.

Computers can suffer crash failures and performance failures. A computer is either crashed or up. Computers can recover from crash failures. Performance failures are defined with the help of a time constant σ . A process of computer e can request to be awakened at local time T to execute some procedure. This process of e is never awakened before T to execute the procedure. This can be enforced by checking that $H_e \geq T$. Computer e suffers a performance failure if the process does not finish executing the procedure by local time $T + \sigma$.

Computers can communicate by exchanging messages. Messages can suffer omission and performance failures: 1) an omission failure occurs if a message is never delivered, and 2) a performance failure occurs if the message is delivered but the transmission delay was greater than some known constant δ .

In summary, the failure model of the timed model considers crash and performance failures of computers and performance and omission failures of messages. The timed model does not bound the failure frequency. For example, there is no guarantee that any message is delivered within δ time units. However, typically δ and σ are chosen such that performance

failures occur infrequently.

Hardware Watchdogs

In this paper, we extend the timed model by a hardware watchdog. Conceptually, the interface of a hardware watchdog consists of a single register that contains a threshold value. A hardware watchdog resets its host computer if the hardware clock reaches the given threshold. Unless this threshold is periodically increased, the computer will crash.

We represent the watchdog of a computer e by W_e and denote the threshold value of W_e at real time t by W_e^t . Let predicate $crashed_e^t$ be true if and only if computer e is crashed at real time t . Formally, the watchdog makes sure that e is crashed whenever the hardware clock H_e shows a value that is greater than or equal to the threshold value: $\forall e, \forall t : H_e^t \geq W_e^t \rightarrow crashed_e^t$.

A computer can recover from a crash. When a computer e boots up at time t_0 , the boot time is stored in a variable $Start_e$: $Start_e := H_e^{t_0}$. When a computer e boots up, we assume its watchdog is set to value $Start_e + \sigma$: $W_e^{t_0} := Start_e + \sigma$. Therefore, the computer has to update its watchdog within σ ticks to stay up.

VII. PROTOCOL

We propose a protocol **PTP** that implements a timed-perfect failure detector in timed asynchronous systems with hardware watchdogs. The proposed protocol is symmetric, i.e., all protocol participants execute the same protocol code. There are three protocol participants: c , d , and e . Each participant can only detect the crashes of the other two participants.

Protocol **PTP** is designed for applications where a computer d has to detect the crashes of only a very small number of computers. If a computer d has to be able to detect the crash of more than 2 computers, d can execute multiple instances of this protocol: each instance has its own “software watchdog” and a computer is crashed by the hardware watchdog as soon as a software watchdog expires. While we are only interested in systems with a small numbers of computers, we discuss in Section VIII-D how the protocol can be extended for a larger number of computers.

A. Overview

The protocol is lease [13] based. The protocol participants grant each other leases to update their watchdogs. Whenever a participant c gets a lease, it sets its watchdog to the end of its new lease term. If a participant c cannot renew its lease, its hardware watchdog will crash c at the end of c ’s lease term. As long as c can renew its lease with either d or e it can stay up. For example, if e crashes, c and d can stay up by granting each other leases. If c becomes “disconnected”, it cannot renew its lease and it will crash. This is needed to enforce the accuracy of the failure detector and it is a useful behavior in systems that shared physical or logical resources (see Section III).

If computer c crashes, it cannot extend its lease, and if c fails to extend its lease, it crashes. To detect the crash of a

computer c , it is sufficient that the other participants d and e detect that c does not have a lease. This detection is done with a distributed snapshot algorithm (e.g., see [6]): the algorithm has to determine if there was a point in time in which c did not have a lease. To perform this snapshot, computer d asks at real-time s if e ’s lease for c has expired (see Figure 3). Consider that e replies at t that c does not have a lease from e . If d receives this reply at time u and if d has not extended c ’s lease since s , d learns that c is crashed at time $t \in [s, u]$ since c does not have a lease at t .

A non-crashed computer detects crash failures within a known amount of time. The messages of the snapshot algorithm are piggy-backed on the lease messages. If c crashes, d can only get lease extensions from e . Since d has to extend its lease, d will either learn that c has crashed or d will crash itself.

This is the intuition of the protocol. The protocol is a little more complicated because it has to deal with clock drifts and skews, and late and dropped messages.

B. Protocol Scenarios

Before we explain the details of the protocol, we describe some important aspects of the protocol using two scenarios: lease extensions and distributed snapshot. The protocol **PTP** defines (besides others) two constants LT , E , a variable $nextRound$ and arrays lt and $output$ (see Figure 4).

Lease Extensions

The current lease of a computer d expires at time $lt_d[d]$. Variable $nextRound$ contains the next local time when d tries to renew its lease: $nextRound_d = lt_d[d] - E$. In other words, d tries to extend its lease E ticks before its current lease expires.

Computer d tries to extend its lease by sending a *request* message m to c and e (see Figure 2). Computer d asks for a lease extension until local time $lt_d[d] + LT$. If c (or e) receives m , it grants a lease extension until some local time $lt_c[d]$ such that if $H_c^s = lt_c[d]$ then $H_c^s > lt_d[d] + LT$. In other words, c makes sure that when its local clock H_c says that the lease it granted to d has expired, then d ’s clock also says that this lease from c has expired.

If d receives a reply message (from c or e), it increments $lt_d[d]$ by LT and it sets its watchdog to the new value of $lt_d[d]$. The watchdog will make sure that d crashes at the end of d ’s new lease – unless d succeeds to extend its new lease.

Distributed Snapshot

If computer c crashes, its leases from d and e will expire: any lease that d granted to c expires at local time $lt_d[c]$ and the lease that e granted to c expires at local time $lt_e[c]$ (see Figure 3). Computer d only suspects c if it learns that all of c ’s leases have expired.

Consider that d sends a request message to e at local time ST and e sends a reply message at local time RST . This reply message contains two time-stamps: RST and Rlt , where $Rlt = lt_e[c]$.

If d receives e ’s reply message, it learns that (1) d ’s lease for c is still expired if $ST \geq lt_d[c]$, and (2) e ’s lease for c is

each other in a timely fashion [8]) grant each other leases, LT is constrained as follows: $LT \geq 2\delta + \sigma$. (2) A participant can be suspected as soon as its lease expires. The protocol has to make sure that no up computer is suspected. Hence, a lease has to expire at the grantee before it expires at the granter. Hardware clocks can drift apart from each other by up to 2ρ . Since the grantee's lease is up to $LT + E$ ticks (a computer asks up to E ticks before its current lease expires for an extension by LT), D is constrained as follows: $D \geq 2\rho(LT + E)$. (3) A computer requests a lease up to E ticks before its current lease expires. Since a lease extension is exactly LT ticks, E should not be greater than LT . Since it can take up to $2\delta + \sigma$ to extend a lease, we constrain E as follows: $2\delta + \sigma \leq E \leq LT$.

To simplify the following discussions and proofs, we will assume that the maximum drift rate ρ of clocks is negligible. In particular, we do not distinguish between duration expressed in real-time and expressed in ticks. Note however that the protocol is correct for non-negligible drift rates (due to the use of constant D).

Negligible Drift Rate: $\rho \approx 0$.

To simplify the protocol, we make one additional assumption that is not part of the timed asynchronous system model. We assume that if a computer c crashes, it stays crashed for at least $DD > LT + E$ time units. Hence, when a computer reboots it can assume that all leases it has granted have already expired¹.

Downtime: *If computer c is crashed at t , then there exists a $u \in [t - DD, t]$ such that for all $v \in [u, u + DD]$ computer c is crashed at v .*

Naturally, the **Downtime** property is satisfied for all computers that need at least DD ticks to boot the operating system. If DD is larger than the boot time of a computer c , one can introduce a delay into the boot sequence to make sure that the operating system takes at least DD ticks to boot.

Protocol **PTP** implements a failure detector TP . To prove the correctness of protocol **PTP**, i.e., that $TP \in \mathcal{TP}$, we have to define constant DD (detection delay) and the output of a failure detector module. The output $TP_d^t(c)$ of TP_d at time t is defined as $TP_d^t(c) := output_d^t[c]$.

We set the maximum detection delay DD to be

$$DD = 4(LT + D) + \Delta \quad (1)$$

The intuition of this constant is that a crashed computer c has a lease that is at most $LT + E \leq 2LT$ and the other computers start suspecting that c 's lease has expired after at most $2(LT + D) + \Delta$, where Δ is due to the fact that a lease can be granted at most Δ after it is requested. After that a computer d will start a distributed snapshot protocol that terminates in at most $LT + E < 2(LT + D)$ since it will either succeed whenever d is able to extend its lease (or terminate when d 's lease expires due to d 's crash).

Theorem: *Failure detector TP satisfies property **Recovering Accuracy**.*

Proof: All times in this proof are given with respect to the

¹This is actually only needed if the computer has granted a lease to another computer that has not expired before the computer crashed.

```

1  const
   external computer myid, left, right;
   external time Δ, δ, σ, ρ, Start;
   time E:=2δ + σ, LT := E;
5  time D:=2ρ(LT+E), IT:=Start+LT;
   type
   enum State {
     crashed, up, recovering };
   enum Participant {
10  myid, left, right };
   var
   external time W;
   State output[Participant] := {
     up, recovering, recovering };
15  time lt[Participant] := {
     IT, -IT, -IT };
   function TP(Participant c)
20  return output[c];
   function other(Participant p)
   if (p == left)
     return right;
25  else
     return left;
   function updateOutput(Participant O, time ST,
30  time RST, time Rlt)
   if (abs(lt[O]) <= ST)
     if (RST >= Rlt)
       output[O] := crashed;
     else
       output[O] := up;
35
   body
   time nextRound := Start + LT - E;
   W := IT;
   forever wait
40  at nextRound:
     Msg := {request, lt[myid]+LT}
     fa-send Msg to left;
     fa-send Msg to right;
     nextRound:= lt[myid]+LT-E;
45
   on receive {request, T}
     from S @ (ST,RT,UB)
     if (UB <= Δ)
       output[S] := up;
       lt[S]:=max(RT+(T-ST)+D,lt[S]);
       O:=other(S);
       fa-send {reply,O,ST,T,lt[O]} to S;
50
   on receive {reply,O,ST,T,Rlt}
     from S @ (RST,RRt,RUB)
     if (RUB <= Δ)
       output[S] = up;
       updateOutput(O,ST,RST,Rlt);
       if (T > lt[myid])
55  lt[myid] = T;
       W := T;
60

```

Fig. 4. Protocol **PTP** that implements TP in timed systems with watchdogs.

time base defined by H_d . For example, T refers to some real-time t such that $H_d^t = T$. One must show that if a computer d starts at time $Start_d$, then for any participant c , variable $output_d[c]$ will not contain **recovering** after time $Start_d + DD$ as long as d stays up.

If $c = d$, i.e., $c = myid$, this condition is trivially true since variable $output_d[d]$ is set to **up** as long as d is up. Hence, let us assume that $c \in \{left_d, right_d\}$ and $e \in \{left_d, right_d\} - c$. The protocol sets $output_d[c]$ to **recovering** only while initializing the variables (line 14). Thus, it is sufficient to show that d sets $output_d[c]$ to some other value, i.e., **up** or **crashed**, by time $Start_d + DD$.

If d receives a timely lease message from c (line 54), it

sets $output_d[c]$ to **up** (line 57). If d executes line 57 by time $Start_d + DD$ with $S = c$ the theorem is satisfied. Hence, let us assume that: (A1) d does not execute line 57 with $S = c$ within time $[Start_d, Start_d + DD]$. Computer d gets a lease from c by executing line 60 with $S = c$. Line 60 is always executed after line 57. Due to (A1), d does not get a lease from c within time $[Start_d, Start_d + DD]$.

If d receives a timely lease request from c , it sets $output_d[c]$ to **up** (line 49). If d executes line 49 by time $Start_d + DD$ the theorem is satisfied. Hence, let us assume that: (A2) d does not execute line 49 with $S = c$ within time $[Start_d, Start_d + DD]$. In this case, d does not grant c a lease in time $[Start_d, Start_d + DD]$ since all leases are sent out in line 52 which is always executed after line 49.

If d crashes in time $[Start_d, Start_d + DD]$, property **Recovering Accuracy** is satisfied. Hence, let us assume that (A3) d is up in $[Start_d, Start_d + DD]$. One must show that in this case, d will set $output_d[c]$ by time $Start_d + DD$ to **crashed** or **up** using function $updateOutput$.

The watchdog W_d of d is set to $Start_d + \sigma$ when d starts (see Section VI). Because we assume that d does not crash (A3), the protocol sets W_d to $Start_d + LT$ by time $Start_d + \sigma$ (line 38). A computer asks for a lease extension of exactly LT ticks (line 41). Since $DD > 2LT$ and due to (A1) and (A3), d has to receive new leases from e by times $Start_d + LT$ and $Start_d + 2LT$. Hence, d executes function $UpdateOutput$ at least twice by time $Start_d + DD$ with parameter c . Computer d schedules the second lease request message for time $Start_d + 2LT - E$. Hence, the send time stamp ST of this message is at least $Start_d + 2LT - E$, i.e., $ST \geq Start_d + 2LT - E$. Since $IT = Start_d + LT$ (line 5) and $LT \geq E$, condition $ST \geq Start_d + IT$ holds. Due to (A2), d does not increase $lt[c]$ beyond IT by time $Start_d + DD$. The second time d executes function $UpdateOutput$, the if condition in line 30 is true. Therefore, d sets $output_d[c]$ to either **up** or **crashed**. \square

Theorem: *Failure detector TP satisfies property **Up Accuracy**.*

Proof: All times in this proof are given with respect to the time base defined by H_d . One has to show that if $output_d^T[c] = \mathbf{up}$, then c has been up at some time in $[T - DD, T]$. To prove this, it is sufficient to show: (L1) if c is crashed in $[T - DD, T]$, then $output_d^T[c] \neq \mathbf{up}$. The output TP_d of a crashed computer d is undefined. Hence, (L1) is true if d is crashed at T . Let us assume that: (A1) $c \neq d$, and that c is crashed in $[T - DD, T]$, and that d is not crashed at T . Note, whenever (A1) is invalid, (L1) is true.

The protocol drops all messages with transmission delays greater than Δ (lines 48 and 56). Hence: (L2) any message received from c in time $[T - DD + \Delta, T]$ is dropped. This means, any message from c with a receive time stamp $\geq T - DD + \Delta$ is dropped.

The lease of a grantee expires after at most $LT + E$ ticks. Hence, if d receives a message from c with an upper bound $UB \leq \Delta$ (lines 46 and 54), within $LT + E$ ticks d either

crashes or sets $output_d[c]$ to **up** (lines 49 and 57). Therefore: (L3) d will not set $output_d[c]$ to **up** using the assignments in lines 49 and 57 within interval $[T - DD + \Delta + (LT + E), T]$.

A grantee's lease term is at most $LT + E + D$ ticks long. Due to (L2), d 's and e 's leases for c expire no later than $B := (T - DD) + \Delta + (LT + E + 2D)$. If d recovers during $(B, T]$, d will initially set $output_d[c]$ to **recovering**. Due to (L3), d can only change $output_d[c]$ by executing function $updateOutput$. Since e 's lease has expired by B , d will not execute line 34.

If d crashes within $[T - DD, T]$, it will not recover by time T (downtime property). Let us assume that: (A2) d is up within $[B, T]$. Since $T - B > LT + E$ and d is up until T , d has to extend its lease from e at least once in $(B, T]$, i.e., d sends a request message after time B (lines 42, 43) and will execute 61 no later than T . Therefore, d also executes function $updateOutput$ within $(B, T]$. Since before d sends its request all of c 's leases have already expired, the two if-conditions in $updateOutput$ hold (lines 30 and 31). Therefore, d will set $output_d[c]$ to **crashed** no later than T . Therefore, (L1) and the theorem are valid. \square

Theorem: *Failure detector TP satisfies property **Crash Accuracy**.*

Proof: All times in this proof are given with respect to the time base defined by H_d . One has to show that if $output_d^T[c] = \mathbf{crashed}$, then c has been crashed at some time in $[T - DD, T]$. To prove this, it is sufficient to show: (L1) if c is up in $[T - DD, T]$, then $output_d^T[c] \neq \mathbf{crashed}$. (L1) is trivially true if d is crashed at T , or if $d = c$. Hence, let us assume that: (A1) $c \neq d$, and c is up in $[T - DD, T]$, and d is up at T .

A computer has to have a lease whenever it is up for more than LT ticks. Since c is assumed to be up in $[T - DD, T]$, c has to have a lease at all points in $[T - DD + LT, T]$. Since $DD > LT$, c has to have a lease at time T . If c got this lease from d (line 52), d has set $output_d[c]$ to **up** (line 49) and $lt_d[c]$ to a value $\geq T$. $output_d[c]$ can only be set to **crashed** in function $updateOutput$ (line 32). Any call to this function during $[T - DD, T]$ will have a send-time stamp $ST < T$: (1) a round-trip of a message takes at least one tick, and (2) the message is received no later than T . Thus, line 32 cannot be executed due to the if-condition in line 30 and because $lt[O] \geq T > ST$.

Hence, let us assume that: (A2) $lt[O] < T$. Let LL be the maximum time such that $LL \in [Start_d^T, T]$ and d executes at LL line 32 or 34. These lines are only executed when d receives a reply message from a computer S . There are three cases:

- **Case 1:** There exists no such LL . Thus, $output_d^T[c] \neq \mathbf{crashed}$ since d can only set $output_d[c]$ to **crashed** in line 32.
- **Case 2:** There exists an LL and $S = c$. In other words, d got its lease from c and hence, had set $output_d[c]$ to **up** in line 57.
- **Case 3:** There exists an LL and $S = e$. In this case, d 's lease for c has already expired since $abs(lt[c]) \leq ST$ (line 30). If $ST \geq T - DD + LT$, c has to have a lease from e

since it is up according to (A1). In this case if-condition 31 is false and hence, $output_d[c]$ is set to **up**.

Let us consider that $ST < T - DD + LT$. A lease expires after at most $LT + E$ ticks. d has to get a new lease by $ST + (LT + E) < T - DD + 2LT + E < T$ since d stays up until T due to (A1). If d gets the next lease from c , it sets $output_d[c]$ to **up** in line 57 by time $ST + (LT + E)$. Note that it cannot change $output_d[c]$ to **crashed** before T due to the selection of LL .

If d gets the next lease from d , $abs(lt[c]) > ST$ (line 30) due to the selection of LL . Hence, this means that d has granted c a lease in line 52. Hence, d has set $output_d[c]$ to **up** in line 49. Since LL was the last time that d were able to change $output_d[c]$ to **crashed**, $output_d[c]$ is **up** at time T . \square

E. Non-Trivial Solution

Failure detector TP is a non-trivial timed-perfect failure detector. Hardware watchdogs make it possible to crash a computer within a bounded time. Hence, one could implement a trivial timed-perfect failure detector that crashes a computer as soon as it starts. This implementation would not update the watchdog and the watchdog would make sure that the computer crashes exactly σ ticks after starting.

TP is a non-trivial timed-perfect failure detector. It crashes a computer only if more than one failure has occurred (e.g., a performance failure and a crash failure). As long as each of the three computers is connected to at least one other computer, no computer is forced to crash. If computer c is connected to d or e , c can renew its lease. Hence, c 's watchdog does not crash c . If one of the three computers crashes, the other two computers can stay up as long as they remain connected.

VIII. MEASUREMENTS

We implemented the protocol in C++. This implementation runs on Linux, Sun, and IRIX. The measurements were done with “defused” watchdogs that were not able to crash a computer. However, we also ran the protocol for more than 1.5 million seconds with a (non-defused) hardware watchdog without experiencing any forced crashes.

In Linux we use a (slightly modified) device `/dev/watchdog` to implement a fully functional watchdog. This device uses either a hardware board or software emulation. The fail-aware datagram service sends messages as UDP packets. It automatically retransmits the last messages (stores one message per type and process) for up to Δ time units to mask message omission failures.

A. Time Constants

We ran the protocol on three of our main center computers: (A) a SGI Challenge which has 4 CPUs and 512 MB of memory. This machine contains home directories and is for general use, (B) a SUN Enterprise 3000 which has 2 CPUs and 256 MB of memory. This machine contains home directories and

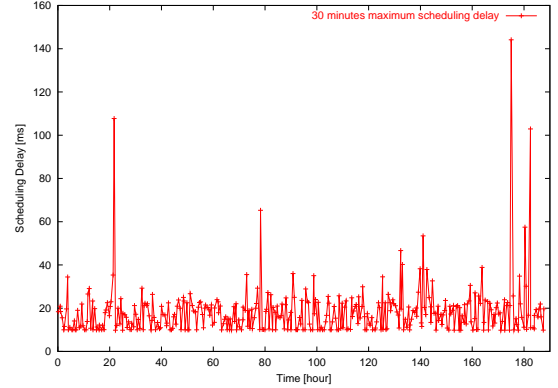


Fig. 5. Timeline of the scheduling delays experienced by protocol **PTP** during a 188 hours interval. The protocol was awakened 338798 times. The maximum scheduling delay was 144ms.

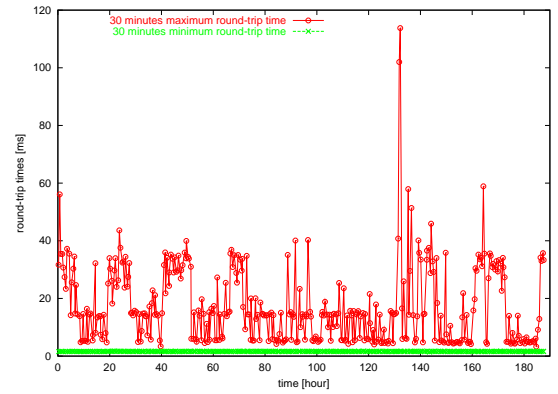


Fig. 6. Timeline of the round-trip times experienced by protocol **PTP** between computers B and C . The protocol measured 338795 round-trips. The minimum time was 555 μ s and the maximum was 118ms.

is for general use, and (C) a SUN Sparcserver 1000 which has 4 CPUs and 262 MB of memory. This machine is for CPU intensive applications.

We measured the scheduling delay SD as the time between

- the local time T at which event $nextRound$ (line 40 in Fig. 4) is executed, and
- the local time S when this event was scheduled to be executed.

Hence, the scheduling delay SD is computed by $SD = T - S$

Figure 5 shows the scheduling delays SD experienced by protocol **PTP** while running on computer B . This measurement covers a period of ≈ 188 hours. The figure shows the variations of scheduling delays over time. In particular, it shows the experienced peak scheduling delays. Note that extended scheduling delays can prevent a computer from renewing its leases. This measurement used the standard round-robin scheduler. Of course, when deploying this protocol one would want to use a real-time scheduler.

Figure 6 shows the round-trip delays experienced by protocol **PTP** running on computer A . If (1) a computer c sends a request message m to computer d at local time ST , and (2) c receives d 's reply at local time RRT , then the round-trip time

is $RRT - ST$.

The fail-aware datagram service retransmits messages to mask omission failures. During the measurement of the round-trip times, a message was retransmitted after $50ms$. However, message omission failures are very rare in our environment. During the depicted 188 hours, only two messages were dropped between computers B and C .

These measurements indicate that for the described computer system the following are reasonable values: $\sigma = 150ms$, $2\Delta = 2\delta = 120ms$. Previous measurements have shown that $\rho = 200\frac{\mu}{s}$ is a reasonable value for unsynchronized and uncalibrated hardware clocks [8]. Hence, one can derive the maximum detection delay as follows:

$$\begin{aligned} LT &= E = 2\delta + \sigma = 270ms \\ D &= 2\rho(LT + E) = 108\mu s \\ DD &= 4(LT + D) + \Delta \approx 1.2s \end{aligned}$$

If there is no specification of the maximum detection delay of crashes in the specification of the application, one can derive LT from the measured round-trip times and scheduling delays. Selecting the above values, a computer is rebooted if it is unresponsive for $\approx 2LT = 540ms$. There are a few caveats when using such a short timeout. For example, some computers “freeze” when a hard-mounted NFS server goes down. This protocol makes sure that a frozen computer is rebooted after approximately $2LT$. On one hand, one can try to make sure that this protocol is not affected by NFS freezes. On the other hand, one can select a larger LT such that a computer is reset if it is frozen for “too long”.

However, all protocol parameters can be derived from the application specification if the latter states a maximum detection times for computer crashes. For example, consider that the specification states that the crash of a computer has to be detected within DD time units. We assume that ρ can be determined from the specification of the hardware clock. We approximate $\Delta \approx LT$ and set $E := LT$. In this case it is reasonable to define LT as:

$$LT := \frac{DD}{5+16\rho}.$$

In summary, all protocol parameters can be derived from the application and the hardware clock specification.

B. Detection Times

To measure the detection times of crashes and recoveries, we ran the protocol on three processes on the same computer. In this way, we simplified the measurement of the detection times. During this measurement, we set $LT = 2sec$, $\rho = 0$, $D = 0$, $E = LT$, and $\delta = \Delta = 2sec$. For these parameters, the protocol guarantees $DD = 10sec$ (see Equation 1).

The time it takes for the protocol to detect that a computer recovers is very short (see Figure 7). This is essentially the time it takes for the recovering process to send its first message to each of the other two participants.

The crash detection times are substantially longer than the recovery detection times (see Figure 7). The reason for this is that a computer c has at any point a lease that is in average $LT + .5LT$ ticks long. (Note that we assume that $E := LT$.)

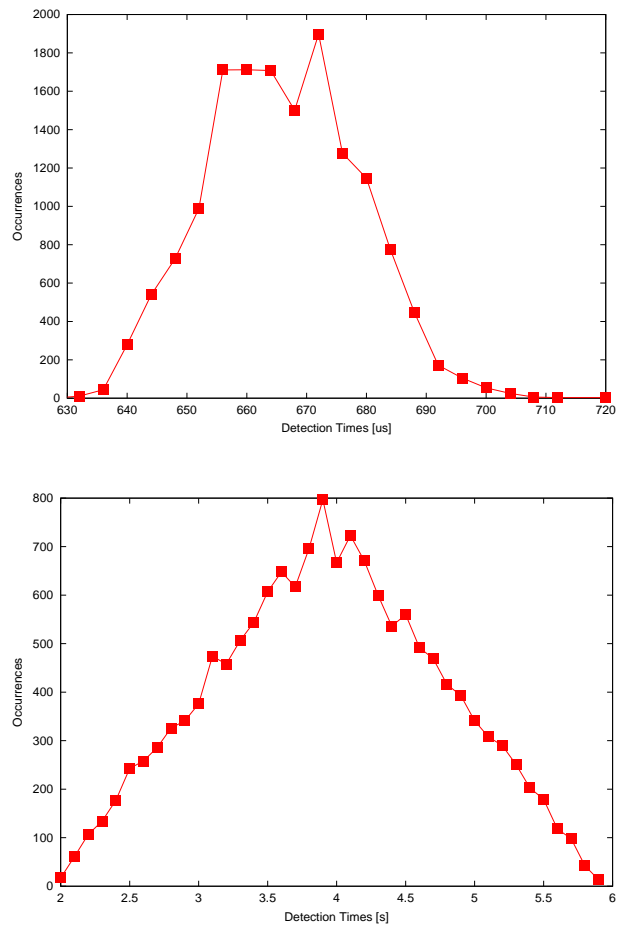


Fig. 7. Measured times until the protocol detects that a computer has recovered (left) and crashed (right). One participant was crashed 15240 times (recovery) and 15043 times (crash), respectively. The minimum recovery detection time is $596\mu s$, the maximum is $9.3ms$, and the average is $677\mu s$. The minimum crash detection time is $2s$, the maximum is $5.9s$, and the average is $3.93s$.

A computer d takes in average about $.5LT$ ticks to request another lease after c 's leases have expired. Hence, one should expect that in average it takes $2LT$ time units to detect a crash failure.

C. Optimization

A slightly more complex protocol can reduce the average crash detection time to about $1.5LT$ ticks. (For $LT \neq E$ this is $E + \frac{LT}{2}$.) The idea is that a computer d runs its distributed snapshot for computer c as soon as its lease for c expires. In the average case the protocol detects a crash of a computer c shortly after c 's lease has expired which is in average about $1.5LT$ ticks.

We measured the performance of this optimization in the same way as in Section VIII-B: $LT = 2sec$, $\rho = 0$, $D = 0$, $E = LT$, and $\delta = \Delta = 2sec$. This measurement confirmed that the average detection time can be reduced to about $1.5LT$ (Figure 8).

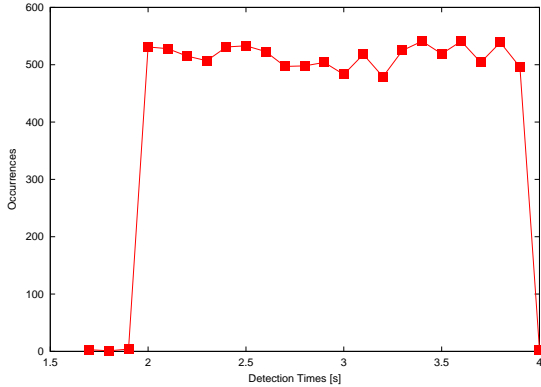


Fig. 8. Measured crash detection times of the optimized protocol. One participant was crashed 10078 times. The minimum detection time is 1.7sec , the maximum detection is 4sec , and the average detection time is 2.95sec .

This optimization does not change the worst case detection time. The additional distributed snapshot is not guaranteed to succeed. For example, the messages of this distributed snapshot might suffer omission failures. Hence, the worst case detection time is determined by the original protocol **PTP**.

D. Extensions

The protocol **PTP** was designed for systems with three computers. This was motivated by the application domains sketched in Section III. For systems with more computers, one can run multiple instances of this protocol as sketched above. Alternatively, one can modify the protocol in the following ways.

Majority Partition Variant

Instead of needing a lease from only one other computer, a computer needs a lease from a majority of the computers to update its watchdog. This majority can include the computer itself, i.e., in a system with three computers a computer needs a lease from one other computer as in protocol **PTP**. This extension ensures that as long as a computer c can communicate in a timely fashion with a majority of computers, c can keep updating its watchdog. The advantage of this extension over **PTP** is that it can tolerate more crash failures.

To detect the crash failure of a computer c , the failure detector has to detect that c does not have leases from a majority of the computers. The detection if the leases of a computer have expired becomes more difficult. For systems with more than 3 computers, the snapshot algorithm has to have two non-overlapping phases. Only if there exists a majority of computers such that their leases granted to a computer c have expired by the end of the first phase and they are still expired at the start of the second phase, computers can suspect c .

Primary Partition Variant

Protocol **PTP** ($F = 1$) and also the above sketched variant needs $2F + 1$ computers to mask F crash failures. This limitation could be overcome by using a primary partition approach similar to that pioneered by ISIS [2]. This can solve the problem with only $\max\{3, F + 1\}$ computers for $F > 0$ (as long

as a majority of the currently not suspected processes survives sufficiently long). The main difference is that a process would need a lease from a majority of the computers that are currently not suspected to be permitted to extend its watchdog. The potential disadvantages of the primary partition approach is that (1) the protocol is more complicated, and (2) that recovery after a complete system failure can in certain instances not be performed in a bounded time even if a majority of the computers are up. The latter is the case, if the members of the last primary partition do not recover after a complete system failure.

IX. PERFECT PROCESS FAILURE DETECTOR AND PROCESS WATCHDOGS

Our protocol **PTP** needs a hardware watchdog to make sure that a suspected computer is crashed before it is suspected. A hardware watchdog kills all processes running on the computer. In some applications it is sufficient to kill one process in a timely fashion instead of crashing the whole computer. For example, if only one process p on a computer e has access to an external shared device, the I/O fencing problem (see Section III) can be solved by crashing only p (and not e) in time. (Note however that often p might actually communicate with a device controller, like a SCSI controller, which is part of the computer and that in turn talks to the external device. In such a case, a hardware watchdog might be needed to make sure the the device controller is crashed in time.)

To implement a perfect *process* failure detector, we can use protocol **PTP** to detecting crashes of the processes participating in the protocol. However, we replace the hardware watchdog by a “process watchdog”. A process watchdog W_p kills the process p no later than specified by the watchdog register. The remainder of this section we explain that some systems permit to implement such a process watchdog in software (in user space).

In most systems one cannot guarantee that a process can execute a task by a given deadline. For example, we measured in [11] the scheduling delays of processes experienced on highly loaded Linux 2.2.14, 450MHz Pentium II system. The maximum measured scheduling delay (i.e., the time by which a task was supposed to be awakened and the time it was actually awakened) was in this measurement 350ms (see Figure 9). Since increasing the load further will increase the scheduling delay further, we cannot assume that we can perform a process to perform critical actions by a given time.

In order to implement a software watchdog we do not have to guarantee that a process performs certain actions by a certain time. Instead we have to make sure that a process p does not perform any instructions between the time p ’s watchdog W_p is supposed to crash p and the time W_p actually crashes p . Note that this is sufficient to solve the I/O fencing problem in case p is the only process that accesses a shared device after it is suspected.

To verify that one can kill a process by a certain time, we implemented a process watchdog using the Unix interval

timer mechanism. We then measured the time between a process executing its last instruction and the time the watchdog is supposed to kill the process (its deadline). Even on a heavily loaded system we were not able to find executions in which a process executed an instruction after its deadline (see Figure 9).

The measurement can be explained as follows. User processes are periodically preempted by the operating system using a timer interrupt. This preemption will happen at least every, say, $10ms$ (time quantum). Often it will happen earlier due to interrupts but never later since the processing of the timer interrupt can only be delayed by another interrupt which will also preempt the process. Before the operating system resumes a process after a preemption, it checks for pending signals for the process. In particular, interval timer signals will be processed before the process is resumed. By setting the interval to at least one time quantum before the deadline, one can make sure that the process does not perform any instruction after its deadline – at least in most “reasonable” operating systems.

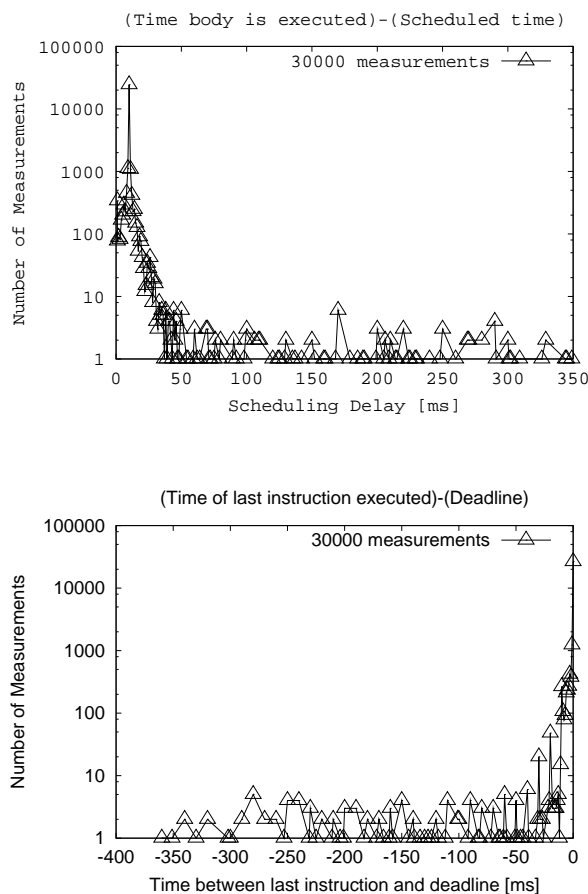


Fig. 9. The scheduling delay experienced by a process might be quite large (left). The body of the signal handler was executed up to $350ms$ after the deadline. However, in no case did a process execute an instruction after the deadline (of its watchdog).

X. RELATED WORK

Failure detectors have received quite some research interest since Chandra, Hadzilacos and Toeg published their seminal paper about the weakest failure detector for solving consensus [3]. Failure detectors were originally defined [4] to augment purely asynchronous systems [12] such that consensus becomes solvable.

Perfect failure detectors are neither implementable in purely asynchronous systems nor in partially synchronous systems [18]. If it were possible to implement then in purely asynchronous systems, one could solve consensus [5] – which is impossible to do in purely asynchronous systems [12]. Perfect failure detection is also impossible to solve in timed asynchronous systems [8]. We circumvent this impossibility by using hardware watchdogs – which are not part of the model defined in [8] – and which permit a timely crashing of computers.

Most related to this work is the paper of Sabel and Marzullo about simulating fail-stop in asynchronous distributed systems [19]. Sabel and Marzullo show how one can simulate a perfect failure detector in a purely asynchronous system. The main idea is to relax the accuracy property by using the happens-before relation: if d suspects c , then this “suspects c event” must not happen before [17] any local event of c : c might execute events after it is suspected but none of these events are “caused” by the “suspects c event”. The happens-before relation is defined such that a crash of c happens before any “suspect c event”. This extension of the happens-before relation makes it non-trivial to simulate a perfect failure detector. The implementation makes sure that c crashes if it is suspected by d .

The main difference between this work and [19] or the work in virtual synchrony [2] is that this work makes sure that a computer is crashed before it is suspected. The protocol of [19] hides the fact that a computer might actually be suspected before it crashes – as long as computers only communicate via some “controlled” message passing mechanism. This is sufficient for applications in which all computers participate in the protocol and there are no hidden channels. If there are hidden channels, e.g., a shared disk, it might become observable that the failure detection is not perfect. In particular, this might lead to inconsistencies and arbitrary failures. The protocol of [19] cannot be used to solve the I/O fencing problem (see Section III).

XI. CONCLUSION

In this paper we addressed the problem of how one can implement a perfect failure detector. The difficulty is to make sure that no computer suspects a non-crashed computer c – even at times when the other computers cannot communicate with c . The main idea is that a computer c commits suicide *before* any other computer is permitted to suspect c . However, a computer is only permitted to commit suicide if more than one failure has occurred. The proposed protocol achieves this using leases and local hardware watchdogs.

If an application specifies worst (or, average case) detection delays, we showed how the parameters of the proposed protocol can be derived from these delays. The approach described in this paper is currently used in several other projects. One project uses the failure detector protocol to rejuvenate computers to fix transient failures and to detect permanent failures in case the rejuvenation does not fix the problem. Another project uses the perfect failure detector to take over the IP and Ethernet address of a failed computer.

Acknowledgments: I would like to thank Matti Hiltunen, Trevor Jim and Karin Högstedt for their very helpful comments and discussions about this work.

REFERENCES

- [1] BHIDE, A., ELNOZAHY, E., AND S.P.MORGAN. A highly available network file server. In *Proceedings of the USENIX Winter Conference* (Jan 1991), USENIX, pp. 199–205.
- [2] BIRMAN, K. P. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth Symposium on Operating System Principles* (December 1985), ACM, pp. 79–86.
- [3] CHANDRA, T., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing* (Aug 1992), pp. 147–158.
- [4] CHANDRA, T., AND TOUEG, S. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing* (Aug 1991), pp. 325–340.
- [5] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43, 2 (March 1996), 225–267.
- [6] CHANDY, M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems* 3, 1 (Feb 1985), 63–75.
- [7] CHERITON, D. R. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, NC, 1993), pp. 44–57.
- [8] CRISTIAN, F., AND FETZER, C. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems* (Jun 1999), 642–657.
- [9] FETZER, C., AND CRISTIAN, F. A fail-aware datagram service. *IEE Proceedings - Software Engineering* 146, 2 (April 1999), 58–74.
- [10] FETZER, C., AND HOGSTEDT, K. Rejuvenation and failure detection in partitionable systems. In *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC 2001)* (Seoul, Korea, Dec. 2001).
- [11] FETZER, C., AND RAYNAL, M. Approximate real-time clocks for scheduled events. In *The 5th IEEE International Symposium on Object-oriented Real-time distributed Computing* (2002, April 2002).
- [12] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32, 2 (Apr 1985), 374–382.
- [13] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Dec 1989), pp. 202–210.
- [14] GUERRAOU, R., OLIVEIRA, R., AND SCHIPER, A. Stubborn communication channels. Tech. Rep. 98-278, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1998.
- [15] HUANG, Y., KINTALA, C., KOLETTIS, N., AND FULTON, N. Software rejuvenation: analysis, module and applications. In *Proc. of the 25th Int. Symposium on Fault-Tolerant Computing* (Pasadena, CA, June 1995), pp. 381–390.
- [16] K. W. PRESLAN, E. A. Scalability and failure recovery in a linux based file system. In *Proceedings of the 4th Annual USENIX Linux Showcase & Conference* (Atlanta, October 2000).
- [17] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM* 21, 7 (Jul 1978), 558–565.
- [18] LARREA, M., FERNNDEZ, A., AND ARVALO, S. On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In *Brief Announcements of the 15th International Symposium on Distributed Computing, DISC 2001* (Oct 2001).
- [19] SABEL, L., AND MARZULLO, K. Simulating fail-stop in asynchronous distributed systems. In *Proceedings 13th Symposium on Reliable Distributed Systems* (Reliable Distributed Systems 1994), pp. 138–147.



Dr. Christof Fetzer is a principal member of technical staff in the dependable distributed systems group at AT&T Labs-Research. He received his PhD from the University of California, San Diego in 1997. Dr. Fetzer has published more than 40 papers in the field of dependable systems. He won two best student paper awards and a Marther Premium from IEE. He has been a program committee member for various conferences and workshops. His current research interests are pervasive dependability and fail-operational system design.