No Time for Asynchrony

Marcos K. Aguilera
Microsoft Research Silicon Valley

Michael Walfish
*University College London, Stanford, UT Austin**

1 Introduction

In their early days, distributed systems were designed under a synchronous (message-passing) model, which assumes bounds on processing and communication delays. These assumptions allowed processes to handle unresponsive processes easily with end-to-end timeouts. However, current distributed systems cannot meet end-to-end timing properties because they are composed of a diverse stack of layers, each with complex timing behavior. At best, these systems have predictable timing in the common case, but even slight deviations from normal load or operating conditions can produce long delays that violate the model's assumptions (§2.1). Since the model fails to represent reality, even systems that are correct under the model can produce errors in practice.

Because of this mismatch, many algorithms and systems have been designed under the *asynchronous model*, in which there are no assumptions on responsiveness and no way to distinguish slow from crashed processes. While this generality is appealing, we argue in this paper that the asynchronous model is poorly suited to building real systems (§2.2): the case people make for this model is problematic, the model can actually be detrimental because it hides useful timing information within the layers of the system, and the model leads to complex systems because of the inability to distinguish slow from crashed processes. Our arguments are supported by the fact that systems that are safe under asynchrony are rare, even when they incorporate components, like Paxos [9], that are specifically designed to tolerate asynchrony (§2.3).

We advocate a different model, namely the asynchronous model augmented with a *perfect failure detector* (PFD) [4], which enables a process to tell whether another process has crashed or is merely slow. This model allows for arbitrary delays, thereby accommodating the timing complexity of current layered systems, yet it avoids a main source of complexity in the asynchronous model: the uncertainty caused by slow processes (§3.1).

Failure detection in systems is often implemented using end-to-end timeouts on heartbeat messages. Such an approach fails to yield a PFD if messages have arbitrary delays (§3.1). We propose a scheme that replaces end-to-end timeouts with a more informed, accurate, and powerful mechanism: *spy modules* or *spies* (§3.2). A spy uses specialized information—including timing assumptions that are local to a layer and reliable—to learn whether a process within a layer is alive. If a process appears crashed, the spy assassinates it and then reports "crashed" with confidence. Spies at many layers form a *spy network* that localizes problems and, if assassination is needed, kills the smallest possible component.

The contributions of this paper are revisiting conventional wisdom about asynchronous systems (§2), advocating a model that departs from pure asynchrony (via perfect failure detectors) as a practical and better alternative (§4), and introducing spy modules as a way to realize this model in practice (§3). We begin our argument by critiquing synchrony and asynchrony in §2; this dilemma highlights the need for a middle ground.

This paper is mostly focused on the scenario of data centers and enterprise networks. However, we think that the ideas are useful more broadly, and in §5 we discuss their extension to wide area networks. Also, this paper assumes that a process or node fails by crashing; what to do in Byzantine environments is future work.

2 Synchrony vs. asynchrony: a dilemma

In the *synchronous model* (or "under synchrony"), there are strict timing guarantees on process execution speeds and communication delays. Thus, a system or process can use the passage of time to infer accurate information. In particular, if a process sends a request to another process but the response does not arrive within a known period, a *timeout* occurs and the first process infers that the second process crashed.

In contrast, the *asynchronous model* is defined by two properties:

A1 There are no timing guarantees, implying processing delays and communication delays are arbitrary;

A2 Crashed and slow processes cannot be distinguished.

Note that A1 alone does not characterize the asynchronous model: there are models where A1 holds but not A2 (e.g., [4]).

In this section, we argue that synchrony and asynchrony are both problematic, creating a dilemma. We also underscore our points about asynchrony with some empirical observations. We describe a proposed resolution to the dilemma in §3.

2.1 Synchrony is dangerous

For a synchronous model to hold, the corresponding timing guarantees must hold in reality. Real-time sys-

*Current affiliation: UT Austin; work done primarily while at UCL.

tems provide these guarantees by constraining load, preallocating resources, controlling the scheduling of tasks, and using special hardware. But outside of the real-time context, synchrony can be dangerous. The reasons are well-known; we briefly review them here.

The main problem is that the synchronous model leads to systems that use end-to-end timeouts. These timeouts are hard to set: end-to-end timing reflects the composition of many layers in the system, including communication layers (i.e., the protocol stack) and computation layers (e.g., threads, processes, virtual machines). And, the timing at each layer often depends on internal factors unknown to the application, such as load, intermittent failures, and background tasks (e.g., garbage collection).

But getting timeouts wrong compromises performance or correctness. If timeouts are too long then, when a process crashes, the system waits longer than necessary, undermining availability. If timeouts are too short (these are called *premature timeouts*) then a process acts on the false premise that the timed-out event will not happen, which can lead to complications, including loss of safety.

2.2 Reconsidering asynchrony

To counter the problems of synchrony, many designers have adopted asynchrony. Indeed, the current conventional wisdom is that to design systems under asynchrony guarantees safety and to do otherwise courts danger. But we think that this view should be reconsidered. We begin with the usual case for the asynchronous model and then argue that this model can actually detract from safety.

At a high level, people design algorithms that are safe under asynchrony because they want a guarantee of safety; the asynchronous model can provide that guarantee because it is general, in two ways. First, the model makes almost no assumptions. Note that it explicitly avoids timing assumptions so avoids the problems reviewed in §2.1. Second, the assumptions that it does make—A1 and A2—are such that if a system is safe under those assumptions, it is also safe if those assumptions are violated. For example, if a system is safe when communication delays are arbitrary, then it is also safe if communication delays are short.

However, we contend that asynchrony can detract from safety. We give three arguments. We begin the first by observing that certain problems, such as consensus [6], do not have solutions that are both safe and live in an asynchronous model. Given this constraint, the conventional wisdom is to take what we call the *async-safety approach*: design components that never violate safety (i.e., they are safe under asynchrony) but are live only in the presence of some synchrony. The trouble with treating liveness as a second-order concern is that, in a real system, the highest layers are typically human users or organizations, which have limited lifespans or time-sensitive goals. In these systems, a loss of liveness at a lower layer can cause a higher layer to miss a deadline, thus violating whole-system safety. (This point has been articulated in the real-time community.)

Our second argument is that, when designing a layer or component under the asynchronous model, designers must expose a narrow interface that hides useful information in that layer or below. In particular, the interface does not allow its users to differentiate slowness from crashes in that layer or below. As a result, when the system is unresponsive, the highest layers (humans, organizations) are forced to guess whether a crash occurred, and incorrect guesses can lead to a loss of safety.

For example, consider a human making a purchase within a Web application. Assume the network and server are asynchronous. If the server is slow, the user can "timeout", incorrectly guess that the server is faulty, and press the "purchase" button a second time. Meanwhile, if the original request had been successful, the result is a repeated purchase, violating safety. Graying out the "purchase" button or tracking duplicate order ids at the server mitigates but does not eliminate this problem because the human can still open a different browser and issue a similar request. The only way to eliminate it is to "freeze" the human, forcing her to wait for confirmation of all requests before issuing any others. Such an unbounded wait causes angry customers and lost business, which are another kind of safety violation.

Our third argument is that asynchrony leads to complex systems, as evidenced by the literature on replication and distributed algorithms. For example, consider Paxos, which is designed to be safe under asynchrony. While expressing the Paxos synod in pseudo-code is simple [10], using it to build a state machine and a system is not [3, 9, 12]. Complexity leads to design flaws or implementation mistakes, which lead to safety violations.

2.3 Empirical observations

We now make two empirical observations to support the points that the asynchronous model is problematic and that a better alternative ought to exist.

The first observation concerns the conventional advice to follow the async-safety approach (defined in §2.2 as upholding safety under asynchrony and achieving liveness only when synchrony holds). We observe that this approach is a custom more honored in the breach than the observance. As a case study, we considered Paxos [9]—a popular consensus algorithm designed under the async-safety approach—and we examined four systems that employ it: Petal [11], FAB [13], Chubby [2], and Consensus Routing [7]. As shown in Figure 1, Petal, Chubby, and Consensus Routing are not safe under asynchrony: they require some synchrony even for safety. FAB ap-

Chubby. Chubby is a replicated lock service that uses Paxos to elect a master replica and to order updates going to each replica. Chubby relies on synchrony, by assuming that no clock has a drift rate beyond an anticipated maximum. If this assumption is violated, Chubby can have two masters. In that case, one of the masters could respond to a client read with stale data, or a client's supposedly coherent cache might contain stale data (a violation of safety).

Petal. Petal is a distributed block storage system that replicates data. It uses Paxos for servers to agree on: which servers are alive (liveness module), a list of disk volumes, and, for each volume, which servers store which blocks. Petal relies on synchrony, by assuming clocks with bounded drift. Without such clocks, the following could happen: (1) a primary server P becomes slow; (2) the liveness module times out on P and declares P dead; (3) as a result, P stops receiving you-are-alive messages; (4) most clients now write to the backup B; (5) however, P remains active (its clock is slow so it does not timeout on the you-are-alive messages), some clients read from P, and they obtain stale data (a violation of safety).

Consensus routing. Consensus routing [7] is a routing protocol that seeks to avoid inconsistent routes, which can cause loops and unnecessary black holes. The protocol proceeds in epochs; in each epoch, nodes use distributed snapshots and Paxos to agree on a set of updates for the next epoch, such that the resulting routes will never be inconsistent. Consensus routing relies on synchrony to ensure that (a) nodes respond to the distributed snapshot protocol on time, and (b) nodes receive the Paxos decision on time. Under complete asynchrony, no useful routing paths are installed, causing the scheme to use a fall-back transient mode in which loops and unnecessary black holes are again possible (a violation of safety).

Figure 1: Use of synchrony for safety in systems that use Paxos.

pears to be safe under asynchrony but devotes considerable complexity to being so.

What can we conclude from these designers' departure from async-safety? One conclusion is that applying this approach in practice is hard. Indeed, for a system to be safe under asynchrony, every component must be safe under asynchrony, yet meeting this requirement, even for a confined component such as a consensus module, necessitates elaborate solutions. Another conclusion is that the generality of "safety under asynchrony" is unnecessary in reality, because true asynchrony does not arise in practice or arises so infrequently that not handling it is a tolerable risk to safety.

Our second observation in this section is that the world is synchronous in a fundamental way. At the sub-atomic level, electrons go around in periodic intervals. At the single component level, CPUs have clocks. And at higher levels, as mentioned above, human users and organizations have deadlines. Given this synchrony, it is not surprising that the systems that we studied above appear to work, even though they are safe only under synchrony.

These two observations suggest to us that the generality of asynchrony might not be needed for safety. But we still have a dilemma. As practiced today, synchrony is problematic because end-to-end timeouts are ad-hoc and dangerous (§2.1). Yet the alternative, asynchrony, causes harm because a process is allowed to block indefinitely without other processes knowing that the process has blocked indefinitely (§2.2). To resolve this dilemma, the next section proposes a principled way to practice "safe synchrony". Note that relying on synchrony for safety, though against the current conventional wisdom, is consistent with our empirical observations above.

3 A middle ground

We advocate an asynchronous model augmented with *perfect failure detectors (PFDs)*, described in §3.1. PFDs have been proposed before [4], but conventional wisdom is that they are impossible to implement in reality. We argue otherwise, by proposing an implementation in §3.2. We describe the benefits of the approach in §4.

3.1 Perfect failure detectors

A PFD is an oracle. A query includes a process name; the PFD's response is whether the process is crashed or alive, where a "process" could be a thread, OS process, machine, etc. The oracle never says that an alive process is crashed, but if there is a crash, the oracle will report the crash after some *detection time*. Our design of PFDs strives to keep the detection time as small as possible. PFDs give a way out of the dilemma above because they eliminate premature timeouts (they never report a slow process as dead) yet do not require other processes to wait forever (they report crashed processes as such).

A model without end-to-end timing assumptions but with PFDs keeps A1 but negates A2; we call this the *PFD model*. Our goal in this section is to realize this model, which requires realizing a PFD, which is challenging.

Previous PFD implementations use end-to-end timeouts on heartbeat messages plus some scheme to avoid failure detector mistakes due to slow messages. One scheme is to run the PFD on a synchronous subsystem [14], but this requires a synchronous network, which may be impractical or costly. Another scheme is to use watchdogs and process suicide [5]. Here, picking the right timeout is hard: small timeouts cause frequent suicides, and large timeouts cause large detection times; in both cases, availability suffers.

3.2 Realizing the PFD model: spies

Given the problems just mentioned, we seek a PFD implementation that does not incorporate end-to-end timeouts. Our scheme is based on four key ideas:

(1) Insider information can sometimes indicate crashes without using timeouts. For example, if a process has disappeared from the OS's process table then we know that it has died. Or if an important thread has exited, the process is effectively dead. To identify such failures may require implementation-specific knowledge. To encapsulate such knowledge, we propose to use tailored modules, called *spies*, that peek at a layer's internal state to tell if a process within the layer has crashed. For example, an OS-level spy views the kernel's process table, and a process-level spy observes an application's key threads.

(2) Use local, not end-to-end, timeouts. As an analogy, consider a supermarket's checkout service. A customer who cannot see the cashier cannot distinguish slow service from "crashed" service. To address this problem, the usual approach is to assume that each customer takes at most T time and the line length is at most N, and to guess that the cashier is dead after a timeout of $N \times T$. Our approach is for the customer to use information internal to the service: our customer would examine the cashier to determine if he is alive, taking his pulse and using timing assumptions particular to the human body (e.g., no pulse for 10 minutes means death). Such internal timeouts are more reliable than end-to-end ones because the latter must reflect the composed behavior of many layers. Internal timeouts may be service-specific; that is why we encapsulate them within service-specialized spies.

(3) Kill if necessary. A spy may not know all the implementation details of a service, and sometimes it may think that a process is crashed without being sure. In these cases, the spy assassinates the process, to ensure that it is indicating a real crash. Assassination has been proposed before (e.g., [5]), but it was coarse-grained and indiscriminate. As we explain below, our strikes are "surgical": we kill only what is necessary, when necessary.

(4) Infiltrate many layers. Spies are deployed at many scopes and layers, forming a network where broader-scope spies monitor narrower-scope ones. For example, a process-level spy can die with the process, so it would not respond to queries, but the OS-level spy monitors process-level spies. Moreover, this spy network can assassinate components "surgically", to kill the smallest non-responsive component (in contrast with [5]). For example, if an OS is responsive but a process is not, only the process is killed, not the machine. If neither OS nor process is responsive, the OS is killed by a network driver spy. If everything is unresponsive, a spy in the switch disconnects the machine from the network. If this spy is unresponsive, there is a network partition and the PFD blocks until the spy responds.

Spies are deployed at the following layers:

- A spy in the network switch tells whether there is link-layer communication between the switch and the machine's network card. It can assassinate by logically disconnecting a network card from the network.
- A spy in the network card responds to a special packet; the spy says whether the network driver is running (e.g., by saying whether past interrupts were serviced). It can assassinate the network driver by powering down the machine (e.g., with lights-out management).
- A spy in the network driver has some knowledge of the life signs of a working OS. It gives its diagnosis in response to a new ICMP message type. It can assassinate the OS by rebooting it.
- A spy in the operating system tells whether a given process is dead, say by looking at the process table in the kernel. It can assassinate the process.
- A spy within a process is a thread that responds to OS signals. The spy determines if the other threads are alive using application-specific knowledge. For example, threads can post their state and progress in shared variables. The spy can assassinate the process.

These spies collaborate to implement a PFD, as follows. Suppose process pA in machine mA wants to know if process pB in machine mB is alive. The PFD in pA tries to talk to the process spy in pB, as follows: pA sends a TCP message to a helper process on mB, which then sends a Unix signal to the spy within pB, waits for a response, and sends it back via TCP to the PFD in pA. Because the process spy in pB may not respond (e.g., if pB is dead), the PFD in pA, simultaneous with the actions above, queries the OS spy on mB, by sending a TCP message to the kernel of mB, asking if process pB is alive.

If the kernel says that process pB is dead, the PFD outputs this information. Otherwise, the PFD in pA keeps querying the OS spy until the OS spy says that process pB is dead or the process spy in pB responds. Because the OS spy on mB may not respond (if mB has crashed), the PFD in pA simultaneously queries the network driver spy on mB, asking if the OS on mB is alive. (For efficiency, it can start this query after a moment, to give a chance for the process spy to respond, so the PFD does not query all spies at all levels simultaneously.) Again, these queries continue until either the network driver spy says the OS on mB is dead or the process spy responds. This continues down the spy network until the last spy (at the network switch). If this spy is unresponsive, there is a network partition and the PFD blocks until it responds.

Summary. Our goal is to eliminate assumption A2, by building a PFD, while keeping A1. One could certainly build a PFD via indiscriminate assassination. However, with such an approach, many processes would die needlessly, so availability would suffer. Instead, we propose

a network of spies. This network can observe the system internals—including internal timing—at many levels to reveal failures and, if necessary, perform surgical assassinations that affect the smallest possible component.

4 Benefits

We now briefly review the benefits of the PFD model (defined in §3.1), which are well-known. The novelty is only that the model is practical so the benefits are achievable.

Like the asynchronous model, the PFD model is fairly general and has broad coverage because it does not rely on end-to-end timing assumptions. Like synchronous models, the PFD model allows applications to know for sure when a process crashed, which is useful because slow processes that appear to be crashed are a source of complexity, inefficiency, and loss of availability.

Here are some concrete examples of the benefits:

— *No need for end-to-end timeout values.* As discussed in §2.1, finding the right timeout value is often hard.

— *Good availability.* With PFDs, processes often learn about a crash quickly so fail-over the system quickly. For example, if an OS spy sees that a process was killed, it reports the process as crashed immediately.

— *Simple fail-over protocols.* Because PFDs never report a crash inaccurately, the protocols for fail-over are simple (fail-over protocols are complex only when they need to handle the case that a component has taken over from a still-working component).

— *Cheap and simple replication.* As an example of the previous point, PFDs enable replication with a simple approach—primary-backup, where the PFD indicates when the backup should replace the primary—instead of schemes based on consensus protocols, which are complex, as mentioned in §2.3.

— *Simple and live distributed algorithms.* Many systems rely on algorithms for atomic broadcast, atomic commit, and leader election. These algorithms often use complex techniques to handle slow processes that appear crashed. PFDs obviate these techniques.

5 Critique and conclusion

One critique of our approach is that deploying spies requires access to system infrastructure (software, operating systems, routers, etc.). Another critique is that, under network partitions, our proposed PFD blocks at the network-switch spy until the partition heals. We think that neither is a major concern in the data center and enterprise environments that we initially target. First, a central entity controls the infrastructure. Second, partitions within a data center are rare and, when they occur, the system is likely to block anyway because a critical service (e.g., a file server) becomes unreachable.

In the wide area, these concerns are graver. To address them, we would need spies in Internet routers and switches. Here, the path to deployment will be slower, requiring standardization efforts. However, we are hopeful because feedback from the network to end-hosts about network state is consistent with current trends in network architecture (e.g., [1, 8]) and because spies' benefits (§4) could provide the needed incentives.

One way to summarize our work is as follows. The synchronous model is no longer applicable because it is impossible to ensure bounded end-to-end delays in practice. One response is the asynchronous model, defined by properties A1 and A2, but this model is also poorly matched to reality because it is too weak. As a result, many designers implicitly relax A1. That approach is dangerous. We propose to explicitly relax A2. By making our deviation from asynchrony explicit, we can depend on synchrony in a safe, principled way. Doing so leads to simple, safe, and live algorithms.

Acknowledgments

Dahlia Malkhi's observation that synchrony is appropriate in some data centers partly inspired this work. Excellent comments and critiques by Lorenzo Alvisi, Allen Clement, Russ Cox, Mike Dahlin, Rachid Guerraoui, Josh Leners, Kyle Jamieson, Don Porter, Swati Rallapalli, Srinath Setty, Sam Toueg, and the anonymous reviewers sharpened and improved this paper. The second author thanks Brad Karp and University College London for their sponsorship.

References

[1] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *HotNets*, Nov. 2004.

[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, Dec. 2006.

[3] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, pages 398–407, Aug. 2007.

[4] T. D. Chandra and S. Toueg. Unrealiable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, Mar. 1996.

[5] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. on Computers*, 52:99–112, Feb. 2003.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, Apr. 1985.

[7] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus routing: The Internet as a distributed system. In *NSDI*, pages 351–364, Apr. 2008.

[8] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, pages 89–102, Aug. 2002.

[9] L. Lamport. The part-time parliament. *TOCS*, 16(2):133–169, May 1998.

[10] L. Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58, Dec. 2001.

[11] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, pages 84–92, Dec. 1996.

[12] D. Mazières. Paxos made practical. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, as of April 2009.

[13] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *ASPLOS*, pages 48–58, Oct. 2004.

[14] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Trans. on Computers*, 51(8):916–930, Aug. 2002.