The Synchronous Data Center

Tian Yang
University of Pennsylvania

Robert Gifford
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Linh Thi Xuan Phan
University of Pennsylvania

Abstract

Today, distributed systems are typically designed to be largely asynchronous. Designers assume that the network can drop or significantly delay messages at unpredictable times, that there is no way to know how quickly a node might process a message, or how soon it might respond, and that the clocks of different nodes are at most loosely synchronized. These assumptions are certainly safe, but they come at a price: many applications really do need predictable performance, which, on top of an asynchronous system, has to be approximated at great cost and with lots of redundancy, and many distributed protocols for asynchronous systems are much more complex and expensive than their synchronous counterparts.

The goal of this paper is to start a discussion about whether the asynchronous model is (still) the right choice for most distributed systems, especially ones that belong to a single administrative domain. We argue that 1) by using ideas from the CPS domain, it is technically feasible to build datacenter-scale systems that are fully synchronous, and that 2) such systems would have several interesting advantages over current designs.

1 Introduction

Today, distributed systems are usually designed to be *asynchronous.* Designers assume that 1) there is no way to predict, or even bound, the time it can take to transmit a message from one node to another; that 2) the network can drop messages at any time; that 3) nodes could take an almost arbitrary amount of time to process messages and respond to them; and that 4) the clocks on different nodes are at most weakly synchronized.

These assumptions seem entirely reasonable and prudent: after all, packets really *can* encounter congestion, routing loops, and various other kinds of trouble in the network that can cause it to be delayed or dropped; with commodity OSes, a node's response time really *can* vary considerably, depending on its workload; and clock-synchronization protocols such as NTP really *do* have limited precision. And even if operators carefully orchestrate services to avoid most of these complications during normal operation, a hardware or software fault can easily disrupt the system's complex choreography.

However, asynchronous designs come at a considerable cost, for at least two reasons. The first is that many applications do require some degree of timeliness: for instance, VoIP services [34], web applications [10, 40], and many other user-facing services all suffer considerably from high delays. A number of mitigations are available, ranging from simple overprovisioning to far more sophisticated approaches [12, 23, 36], but most of them have a substantial cost and/or are merely best-effort – that is, they cannot guarantee any delay bound.

The second reason is that asynchrony has *inherent* costs for algorithm design. For instance, an asynchronous protocol can never be sure that a message will *not* arrive – no matter how long a node waits, the message could always have been delayed for exactly that long and could be just about to arrive. Many developers resort to timeouts to handle this problem, but these are notoriously hard to choose [43] and can cause rare and complex failure modes [18, 24], which has given rise to an entire line of work on failure detectors [11, 32]. In contrast, protocols for synchronous systems are often cheaper and much simpler; for instance, the lower bound for BFT is $3f + 1$ in asynchronous systems but only $2f + 1$ in synchronous systems [30].

Much of the literature seems to have accepted these costs as inevitable – the "cost of doing business" in distributed systems, as it were. The goal of this paper is to question that assertion. We argue that, in fact, *most of the asynchrony in today's distributed systems is avoidable*, and that it would be both possible and desirable to build datacenter-scale systems that are (almost) entirely synchronous. Our optimism is based on a combination of three factors: 1) recent advances in datacenter hardware, such as RDMA and SSDs; 2) recent progress on the research side, including systems like Fastpass [36] and DTP [31]; and 3) insights from other research areas, such as cyber-physical systems (CPS), which have

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotOS '19, May 13–15, 2019, Bertinoro, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321442

been building fully synchronous systems for decades, albeit at much smaller scales.

Building a fully synchronous system would be far from trivial. In a system with multiple administrative domains, it would be nearly impossible, since a given domain cannot be sure what the other domains are running on their nodes. This is why we are initially targeting data centers, which tend to be controlled by a single entity that controls everything, from the hardware on up. But even in a data center, synchrony would be challenging to achieve. A key reason is that, somewhat analogous to security, synchrony is a property that is easy to lose but hard to regain: a single asynchronous component can destroy all end-to-end timing guarantees. Thus, incremental deployment seems out of the question.

We are aware that this paper may seem somewhat heretical – it proposes to flip a best-practice approach on its head; it flies in the face of the end-to-end principle; and it proposes sweeping changes to system designs that, at first glance, may look like an operator's worst nightmare. Despite all this, we believe that it makes sense to have a discussion about this approach. We believe that the technology is ready for a change, and that there are considerable benefits to be had.

2 The case for synchrony

Asynchrony is an *assumption* about a distributed system. In its typical form, it says that a) nothing is known for certain about the relative speeds of processes or about the delay in delivering a message, and that b) nodes do not have access to synchronized clocks. Notice that this "pure" definition does not assume *any* bound on delays *at all*; thus, a protocol that uses this assumption must work correctly even if, for instance, a message takes thousands of years to deliver. There are weaker definitions, such as partial synchrony [19], which assumes that there is some upper bound on both the message delay and the relative processing speed of the nodes, but protocols with this assumption are less common in practice.

A related assumption is that the network is *unreliable* – in other words, that any transmitted message can be lost. In its strongest form, this assumption means that very little can be done, since the network may very well lose *all* messages that are sent over it, so a more common assumption is that messages are eventually received if they are retransmitted often enough. However, even the weaker version means that one cannot effectively bound the time it takes to send information from one node to another, since it may need to be retransmitted several times.

2.1 Is asynchrony inherent?

No real-world system is perfectly synchronous – the laws of physics prevent us from measuring time *exactly*, no pair of real-world clocks runs at *exactly* the same speed, and, due to platform features such as speculation, one often cannot predict *exactly* how long a given CPU will take to execute a particular instruction. Also, to our knowledge, no real-world network can guarantee that all messages will always be delivered.

But – and this is a critical point! – *these inherent limitations are very small.* Google's Spanner system synchronizes clocks in different data centers to within 4 milliseconds [16]; the execution speed on commodity platforms can be stabilized to within 1–2% [13], and even the worst links in real-world data centers have a corruption loss rate on the order of 10^{-3} [50]. Even the seemingly inevitable queueing delays are not inherent: systems like Fastpass [36] have demonstrated that even commodity network hardware can be scheduled so precisely that packets never experience queueing delay.

So the asynchrony assumption seems *incredibly conservative*: real-world systems really do have a considerable amount of synchrony, and, by designing protocols for the asynchronous model, we are explicitly declining the opportunity to profit from this synchrony.

2.2 Why is asynchrony so common?

If real-world systems are not inherently asynchronous, the question arises why it is nevertheless so common to assume that they are. We speculate that there are two main reasons:
Administrative domains: If parts of the system are controlled by another entity, it is hard to reason about timing with any certainty, since one cannot know how (and how well) the other domain is handling timing. This is a significant concern, e.g., for the Internet as a whole. However, it is less of a concern for data centers, where the operator can – at least in principle – know, and control, every single aspect of the system.
Lack of predictability: To get complete synchrony, every part of the system has to support it; even a single component with unpredictable timing (such as a commodity kernel) can destroy the end-to-end guarantees, since it will be hard to tell what the worst case is. Since much of the current off-the-shelf hardware and software is asynchronous, it is easy for designers to throw up their hands and to dismiss synchrony as unrealistic from the get-go. This concern, too, is significant but not insurmountable: the current situation is akin to a vicious circle, and it may be possible to break out of it simply by building an initial stack of synchronous components.

2.3 What is so bad about asynchrony?

Even if it were somehow feasible to build large synchronous systems, the reader may be wondering why that might be worth the effort. After all, current data centers seem to be working just fine – so why would we want to turn the design on its head?
Source of common problems: One answer is that asynchrony is at the heart of many complex problems that the community has been dealing with. For instance, considerable effort has been spent on topics such as mitigating congestion [36, 48], reining in tail latencies [12, 38, 42], smart load

balancing [17, 23] and performance debugging [6, 7, 9, 14, 27, 49]; in a hypothetical synchronous data center where all activity is carefully scheduled, many of these problems could potentially be alleviated or even disappear completely.

Not as simple as it appears: Another reason is that asynchronous designs are more complex than they intuitively seem. For instance, most real-world applications cannot wait for a response indefinitely and thus must eventually time out requests – but timeouts are notoriously hard to set correctly [2, 43] and can lead to difficult failure modes [18, 24] that then have to be dealt with at considerable expense. Provisioning buffers and choosing transmission rates are difficult if the network conditions are not known and must be laboriously inferred from RTTs. Etc. The complexity of these all-too-familiar issues must be weighed against the perceived additional complexity of a synchronous design.

Problematic for security: A third reason is that asynchrony can be problematic for security: if the system does not reason about how long operations can/should take, it is easy to accidentally create a vector for DoS attacks or a side channel. One prominent example is the paper by Clement et al. [15] that showed that many fault-tolerance protocols can be used to slow down the system to a crawl, and there is an entire line of research on exploiting various kinds of timing observations to extract secrets, e.g., on cloud platforms [39]. A synchronous design would not instantly cure all of these problems, but, by providing a way to at least reason about timing, it could make it much easier to deal with them.

Inherently higher cost: Finally, asynchrony is known to have an *inherent* cost that has been studied, e.g., by the theory community. It is known that many distributed tasks either have worse lower bounds in asynchronous systems [3, 5] or cannot be performed at all when nodes can fail [21]. For instance, BFT has a widely known lower bound of $3f + 1$ nodes in an asynchronous setting, but even Lamport's original paper contained a protocol with $2f + 1$ nodes for the synchronous setting [30]. This lower complexity, for agreement and other types of protocols, could mean better performance for existing systems, and/or access to stronger protocols (such as BFT) that are currently seen as too expensive.

We note that the synchronous variant of BFT, like many other fully synchronous protocols, is not as widely known. We suspect that this is partly because the protocol is much simpler (some might call it "trivial") and not as interesting. But when it comes to building systems, simplicity is good – and triviality is even better!

3 How could it be done?

A key design principle for a synchronous data center would be that *everything is scheduled*. Intuitively, the data center would have a giant timetable that specifies exactly which resources are used when and for what – for instance, which computations should be run on which nodes, and at what times, or when packets should be sent, and over which links. In practice, this schedule would of course not be maintained in a single place, to avoid bottlenecks and single points of failure, and no node would need to know every single detail. The important thing is that nodes can no longer be allowed to use resources spontaneously – e.g., for internal housekeeping tasks – since this might disrupt some other activity elsewhere in the data center of which they were not aware.

A second design principle would be that *resources must be part of all APIs*: components would need to explicitly specify the "cost" of each operation (e.g., in terms of CPU time or bandwidth), and the caller would need to allocate the necessary resources and "charge" the invocation to that allocation. This seems unavoidable: a component in the middle of the protocol stack can only give meaningful timing guarantees if it can rely on guarantees from the lower layers it relies on.

The third principle would be that *resource allocations are rigorously enforced*: if a component tries to run longer, or send more traffic, than it has requested, it would have to be preempted and would need to request additional resources. This is the only way to reliably enforce isolation, and for components to be sure that the resources it has allocated will actually be available to it.

3.1 Network layer

Whenever a node wants to send a packet over the network, it would – at least conceptually – need to request permission from the scheduler. This may sound absurd to some readers, and until SIGCOMM 2014 it would have sounded absurd to the authors as well, but Fastpass [36] showed that this is not only feasible in theory but actually quite practical: a single machine can schedule more than two Terabits of traffic. Fastpass does not consider real-time scheduling, but it does provide the infrastructure for scheduling networks at packet granularity; we see no reason why it could not be extended to other policies.

As the Fastpass paper already notes, this approach essentially eliminates the need for queueing in the network; this not only makes latencies very predictable, it also eliminates the most common source of packet drops, leaving only the (rare) losses due to packet corruption and component failures. Thus, with a bit of forward error correction, it seems feasible to build a network that is almost completely lossless in the absence of failures.

Once these mechanisms are in place, the higher protocol layers could be simplified considerably, since much of their current complexity (congestion control, flow control, retransmissions, reordering, ...) is needed to deal with the consequences of packet loss, variable delay, and the fact that the available bandwidth is not known. To push this point to the extreme: it is not even completely clear that one would still need sequence numbers, since the receiver can know which packets are supposed to arrive when and from where!

3.2 Synchronized clocks

Since a synchronized data center would rely heavily on time, precise clock synchronization would be a key building block. Clock synchronization has traditionally had a somewhat bad reputation, and the common knowledge is that it is somewhat imprecise. This is certainly true in wide-area networks – but recall that our focus is a data-center network, and moreover, one that has virtually no queueing delays (Section 3.1)! Having stable propagation delays means much higher precision, and, as DTP [31] has shown, one can get all the way down to nanoseconds by enlisting some help from the physical layer.

Still, the laws of physics prevent us from synchronizing clocks *exactly*, but we are not aware of any important use case that requires exact time; a small tolerance is usually okay, as long as it is quantifiable.

3.3 Building blocks

In combination, synchronized clocks, bounded delays, and reliable transmission (in the absence of faults) unlock the distributed-systems equivalent of the land of plenty, where grilled geese fly directly into one's mouth and cooked fish jump out of the water. We mention only two examples. The first is that message ordering, a notorious challenge in many building blocks for distributed systems, becomes almost trivial: if clocks are synchronized to $\pm\tau$ and the maximum transmission delay is d, nodes can simply use their local timestamps as sequence numbers (with node IDs as tiebreakers) and process messages after a short delay of $2\tau + d$, to make sure they have seen all messages from other nodes that need to be ordered before theirs (see, e.g., Spanner [16]). This property alone can potentially help to simplify a wide range of ordering-related protocols that deal with consistency, replication, and group communication.

Our second example is fault detection. In a synchronous system, the *absence* of a message at an expected time carries meaning as well! For instance, if the scheduler has arranged for a task on a node N_1 to send a message to another node N_2 at some time t, then all the necessary resources have been provisioned and the message really should arrive no later than $t + 2\tau$. If the message does not arrive, then something must have gone wrong somewhere, and N_2 can trigger recovery right away. Although the reality will be a bit more complicated (because of FEC, etc.), synchrony can still dramatically simplify fault detection, which is currently a line of research on its own [22, 32].

3.4 Node hardware

To achieve end-to-end synchrony, it is not sufficient if the network is synchronous – the nodes must be as well. Today's commodity platforms do not fit that description: most have been optimized for high throughput, using massive caches and extensive speculation, and, as a result, programs running on these platforms do not have very predictable timing. Other approaches have been investigated in research – e.g., the fully synchronous PRET machine [20], which delivers both high performance and predictable timing – and of course many embedded systems are designed for predictability. But, at least at first glance, current hardware does not have very good support for synchrony.

However, the picture is not quite as bleak as it may seem. Prior work [13] has shown that, even on commodity hardware, timing stability of less than 2% is achievable. SSDs are replacing the more variable HDDs, and commodity CPUs have been adding more and more features that can help with isolation: a recent example is Intel's Cache Allocation Technology (CAT) feature, which can be used to explicitly manage cache usage. Moreover, the recent Meltdown/Spectre disaster [29, 33] has increased skepticism towards speculation and will presumably lead to better hardware support for predictable timing. Even though the primary purpose of these mechanisms is security, we speculate that they could also be used to enforce synchrony.

3.5 Software

Even if the network were synchronous and the hardware had the necessary mechanisms to implement guaranteed timing, it would be up to the OS kernel to actually deliver this guarantee to the applications. Recall that the kernel should ideally a) account for *all* resources, b) make resource usage explicit in *all* APIs, and c) be able to enforce *all* allocations through preemption. Most popular kernels do only a subset of this, and only for a subset of the system resources: for instance, Linux will consume CPU and memory resources for various internal housekeeping tasks, which can cause big latency spikes, and time or memory usage within the kernel itself is not always properly accounted for. Moreover, retrofitting real-time technology into an existing kernel could be difficult because many design decisions have already been made without timing guarantees in mind; thus, it is not reasonable to expect that a kernel like Linux can be upgraded with a short "synchrony patch".

Nevertheless, we see no *fundamental* reason why a suitable kernel cannot be built. An existing real-time OS, or the recently developed real-time Xen hypervisor [47], would already closely approximate our requirements, and the technology for handling the more esoteric resources, such as kernel memory, already exists [8, 25].

On the software side, a key challenge is that applications would need to provide worst-case execution times (WCETs) in order to be scheduled. Fortunately, WCET analysis is an old problem [45] that has been studied in the real-time community for many years, and there is a variety of solutions available, including static analysis, special language support, linear type systems, and profiling-based tools.

3.6 Responding to faults

Faults are a serious threat to a synchronous system because they can disrupt the careful choreography that the scheduler attempts to establish. For instance, the scheduler may direct nodes to send traffic over a certain link, without realizing that the link has failed, or it may assign a task to a node that has crashed. Just like an asynchronous system, a synchronous system needs a way to respond, and ideally to tolerate, such a situation.

At first glance, the same options are available as in asynchronous systems: for instance, the system can accept a brief outage while it reconfigures itself (e.g., brief routing loops while the routing protocol is computing a new set of paths), or it can establish enough redundancy up front (e.g., in the form of redundant paths or task replicas) to mask the symptoms of a limited number of faults. But there is an additional complication: with synchrony, the system will eventually need to transition to a new schedule, and this can easily lead to a brief period of chaos, during which tasks and traffic are being migrated and miss their deadlines as a result. Unless this threat can be averted, there is a risk that the system will become very brittle.

However, synchronous systems could avoid this problem using a trick from the cyber-physical systems literature. CPS can operate in multiple different *modes* – for instance, the control unit in a car might have a mode for manual driving, another for cruise control, and a third for emergency braking. These modes can differ both in the sets of tasks that they run (e.g., the cruise-control task might run only in the first mode, but not in the others), as well as the parameters of these tasks. Since CPS are often safety-critical, they must not miss deadlines even while transitioning between modes; for instance, airbags must be inflated within a very short time from impact, regardless of the current state of the system.

To accomplish this, the CPS literature has developed *mode change protocols (MCPs)* [37], which are essentially small, transient schedules that transform a system from one mode to another – by stopping or migrating old tasks, or by launching new tasks – without missing any deadlines in the process. MCPs can be precomputed offline and then simply executed when a mode transition is triggered; this is somewhat analogous to fast rerouting [44], which similarly precomputes alternative paths that can be activated in the event of a failure. On a grander scale, MCPs could help a synchronous data center to recover from faults without losing its synchrony.

3.7 Scheduling

The elephant in the room, of course, is scheduling: maintaining a detailed schedule for a datacenter-sized system would be a truly herculean task. And, despite the fact that real-time scheduling has been studied for decades, there appears to be relatively little prior work. Most of the classical algorithms were designed for uniprocessors, and even current work tends to focus mostly on multicore CPUs; if networks are considered at all, they tend to be very small.

However, there are at least two reasons to be optimistic. One is the fact that, over the past few years, data center scheduling systems have been moving towards finer and finer granularities: for instance, Sparrow [35] can schedule 100ms tasks on tends of thousands of cores. True, Sparrow does not perform real-time scheduling, but, at the very least, fine-grained scheduling at datacenter scale is not as implausible as it may once have been. The ongoing work on Function-as-a-service (FaaS) systems [1] is going in a similar direction. Second, there are now interesting candidate solutions, such as the *compositional scheduling* approach from the CPS literature [41]. This approach provides a way to recursively and efficiently subdivide a large scheduling problem into smaller ones, and it also makes it possible to (mostly) schedule subsystems independently, which avoids the need for a single global "master schedule" for the entire system.

4 Related Work

Synchronous systems have been studied thoroughly by at least two other communities: the theory community and the real-time/embedded/CPS community. We have already discussed a number of relevant papers from both communities. However, to our knowledge, nobody has gone as far as to suggest that the fully synchronous model can be applied to anything as large as a data center.

Aguilera and Walfish [2] argues that, outside of a real-time context, the synchronous model is dangerous because it often leads to end-to-end timeouts, which are hard to set. It proposes a middle ground that adds an approximation of a perfect failure detector [11] but leaves the unbounded processing and communication delays in place. This approach should be easier to implement than ours, but it provides only a subset of the benefits.

Prior work has shown how to make specific parts of a data center more predictable – including, e.g., SILO [26], Pulsar [4], and Chronos [28] – or even how to support real-time workloads [46]. However, these papers use a more evolutionary approach that is far easier to deploy incrementally but fundamentally cannot lead to the design we are advocating; recall from Section 3.3 that many of the potential benefits would accrue only once the system is *fully* synchronous.

5 Conclusion

The goal of this paper was to question an assumption that is very common in distributed systems today: that systems should be asynchronous by default, and that the synchronous model is impractical and not a realistic option, except for a few special cases (such as embedded systems). We have made a case that this assumption is wrong, and that, by leveraging some ideas from the CPS domain, it may in fact be possible to build fully synchronous datacenter-scale systems.

Algorithms for the synchronous model are often considered "boring", and for a reason: they are often much simpler (and less expensive) than their asynchronous equivalent, to the point of being almost trivial. But, in this particular case, "boring" and "trivial" could be a very good thing!

Acknowledgments

We the anonymous reviewers for their thoughtful comments and suggestions. This work was supported in part by NSF grants CNS-1703936, CNS-1563873, and CNS-1750158.

References

[1] NSF Cloud 3.0 workshop report. http://pages.cs.wisc.edu/~akella/cloud3workshopreport.pdf, Jan. 2018.

[2] M. Aguilera and M. Walfish. No time for asynchrony. In *Proc. HotOS*, 2009.

[3] D. Alistarh, S. Gilbert, R. Guerraoui, and C. Travers. Of choices, failures and asynchrony: The many faces of set agreement. *Algorithmica*, 62(1):595–629, Feb 2012.

[4] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proc. OSDI*, 2014.

[5] E. Arjomandi, M. J. Fischer, and N. A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *J. ACM*, 30(3):449–456, July 1983.

[6] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proc. OSDI*, Oct. 2012.

[7] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. SIGCOMM*, 2007.

[8] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. OSDI*, 1999.

[9] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. OSDI*, 2004.

[10] J. Brutlag. Speed matters for google web search. http://services.google.com/fh/files/blogs/google_delayexp.pdf.

[11] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[12] A. Chatzieleftheriou, S. Legtchenko, H. Williams, and A. Rowstron. Larry: Practical network reconfigurability in the data center. In *Proc. NSDI*, 2018.

[13] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI*, 2014.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. DSN*, 2002.

[15] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. NSDI*, 2009.

[16] J. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. OSDI*, 2012.

[17] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *Proc. INFOCOM*, 2011.

[18] T. Dai, J. He, X. Gu, and S. Lu. Understanding real-world timeout problems in cloud server systems. In *Proc. IC2E*, 2018.

[19] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), Apr. 1988.

[20] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proc. DAC*, 2007.

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

[22] F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, Feb. 2011.

[23] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro load balancing for low-latency data center networks. In *Proc. SIGCOMM*, 2017.

[24] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proc. SoCC*, 2014.

[25] A. Haeberlen and K. Elphinstone. User-level management of kernel memory. In *Proc. ACSAC*, 2003.

[26] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *Proc. SIGCOMM*, 2015.

[27] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proc. SIGCOMM*, Aug. 2009.

[28] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proc. SoCC*, 2012.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. ArXiv 1801.01203, 2018.

[30] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[31] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally synchronized time via datacenter networks. In *Proc. SIGCOMM*, 2016.

[32] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon Spy Network. In *Proc. SOSP*, 2011.

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. ArXiv 1801.01207, 2018.

[34] A. P. Markopoulou, F. A. Tobagi, and M. J. Karam. Assessment of VoIP quality over Internet backbones. In *Proc. INFOCOM*, 2002.

[35] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. SOSP*, 2013.

[36] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proc. SIGCOMM*, 2014.

[37] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.

[38] W. Reda, L. Suresh, M. Canini, and S. Braithwaite. BRB: BetteR Batch scheduling to reduce tail latencies in cloud data stores. In *Proc. SIGCOMM*, 2015.

[39] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proc. CCS*, 2009.

[40] N. Shalom. Amazon found every 100ms of latency cost them 1% in sales. GigaSpaces blog, https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them- 1-in-sales/, 2008.

[41] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS*, 2003.

[42] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. NSDI*, 2015.

[43] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proc. SIGCOMM*, 2009.

[44] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang. R3: Resilient routing reconfiguration. In *Proc. SIGCOMM*, 2010.

[45] R. Wilhelm et al. The worst-case execution-time problem. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[46] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky. RT-OpenStack: CPU resource management for real-time cloud computing. In *Proc. CLOUD*, 2015.

[47] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proc. EMSOFT*, 2011.

[48] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye. DIBS: Just-in-time congestion mitigation for data centers. In *Proc. EuroSys*, 2014.

[49] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI 2: CPU performance isolation for shared compute clusters. In *Proc. EuroSys*, Apr. 2013.

[50] D. Zhuo, M. Ghobadi, R. Mahajan, K. Förster, A. Krishnamurthy, and T. E. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. SIGCOMM*, 2017.