acmqueue

There is No Now

Problems with simultaneity in distributed systems

Justin Sheehy

"Now."

The time elapsed between when I wrote that word and when you read it was at least a couple of weeks. That kind of delay is one that we take for granted and don't even think about in written media.

"Now."

If we were in the same room and instead I spoke aloud, you might have a greater sense of immediacy. You might intuitively feel as if you were hearing the word at exactly the same time that I spoke it. That intuition would be wrong. If, instead of trusting your intuition, you thought about the physics of sound, you would know that time must have elapsed between my speaking and your hearing. The motion of the air, carrying my word, would take time to get from my mouth to your ear.

"Now."

Even if I held up a sign with that word written and we both looked at it, our perception of that image would not happen at the same moment, as the light carrying the information about the sign would take a different amount of time to reach each of us.

While some things about computers are "virtual," they still must operate in the physical world and cannot ignore the challenges of that world. Rear Admiral Grace Hopper (one of the most important pioneers in our field, whose achievements include creating the first compiler) used to illustrate this point by giving each of her students a piece of wire 11.8 inches long, the maximum distance that electricity can travel in one nanosecond. This physical representation of the relationship between information, time, and distance served as a tool for explaining why signals (like my metaphorical sign above) must always and unavoidably take time to arrive at their destinations. Given these delays, it can be difficult to reason about exactly what "now" means in computer systems.

There is nothing theoretically preventing us from agreeing about a common idea of "now," though, if we carefully plan ahead. (Relativity isn't a problem here, though it is a tempting distraction. All of humanity's current computing systems share a close enough frame of reference to make relativistic differences in the perception of time immaterial.) NTP (Network Time Protocol),[14] used for synchronizing the clocks between systems on the Internet, works in part by calculating the time that messages take to travel between hosts. Once that travel time is known, a host can account for it when adjusting its clock to match a more authoritative source. By providing some very precise sources (clocks based on continuous measurement of atomic radiation) in that network, we are able to use NTP to synchronize computers' clocks to within a small margin of error. Each of the satellites making up the worldwide GPS contains multiple of these atomic clocks (so that a single clock failing doesn't make a satellite worthless), and the GPS protocols allow anyone with a receiver—as long as

they can see enough satellites to solve for all of the variables—to determine not only the receiver's own position, but also the time with excellent precision.

We have understood these protocols for a few decades, so it would be easy to believe that we have overcome this class of problems and that we ought to be able to build systems that assume our clocks are synchronized. Atomic clocks, NTP, and GPS satellites provide both the knowledge and the equipment to account for the time it takes for information to travel. Therefore, systems anywhere should be able to agree on a "now" and to share a common, single view of the progress of time. Then whole categories of hard problems in networks and computing would become much easier. If all of the systems that you care about have the exact same perception of time, many of these problems become tractable even when some of the hosts involved are failing. Yet, these problems do still exist, and dealing with them is not only a continuously active area of research, but also a major concern when building practical distributed systems.

You might look at the mature mechanisms available for understanding time and believe that researchers and system builders are doing a huge amount of unnecessary work. Why try to solve problems assuming that clocks may differ when we know how to synchronize? Why not instead just use the right combination of time sources and protocols to make the clocks agree, and move on past those problems? One thing makes this implausible and makes these problems not only important, but also necessary to face head on: everything breaks.

The real problem is not the theoretical notion of information requiring time to be transferred from one place to another. The real problem is that in the physical world in which computing systems reside, components often fail. One of the most common mistakes in building systems—especially, but not only, distributed-computing systems on commodity machines and networks—is assuming an escape from basic physical realities. The speed of light is one such reality, but so is one that is more pernicious and just as universal: we cannot make perfect machines that never break. It is the combination of these realities, of asynchrony and partial failure, that together make building distributed systems a difficult pursuit. If we do not plan and account for failures in individual components, we all but guarantee the failure of combined systems.

One of the most important results in the theory of distributed systems is an impossibility result, showing one of the limits of the ability to build systems that work in a world where things can fail. This is generally referred to as the FLP result, named for its authors, Fischer, Lynch, and Paterson.[8] Their work, which won the 2001 Dijkstra Prize for the most influential paper in distributed computing, showed conclusively that some computational problems that are achievable in a "synchronous" model in which hosts have identical or shared clocks are impossible under a weaker, asynchronous system model. Impossibility results such as this are very important, as they can guide you away from going down a dead-end path when designing your own system. They can also provide a snake-oil detector, so you can feel justified in your suspicion about anyone who claims a product does something you know to be impossible.

A related result, better known than FLP by developers today, is the CAP theorem (for consistency, availability, and partition tolerance). This was first proposed informally by Eric Brewer,[5] and later a formal version of it was proven by Seth Gilbert and Nancy Lynch.[9] From a distributed systems theory point of view, the CAP theorem is less interesting than FLP: a counterexample "beating" the formal version of CAP assumes an even weaker and more adversarial model of the world than FLP, and demands that even more be achieved within that model. While one is not exactly a subproblem

of the other, FLP is a stronger, more interesting, and perhaps more surprising result. A researcher already familiar with FLP might find the CAP idea a bit boring.

It would be reasonable, though, to think that perhaps the value of CAP is to be more approachable and more easily understood by those not steeped in the literature of distributed systems. That would be laudable and worthwhile, but the past several years have shown (through dozens of articles and blog posts, many of which badly misunderstand the basic idea) that the idea of CAP has, sadly, not been an easy way for today's developers to come to terms with the reality of programming in a distributed and imperfect world. That reality, whether from the point of view of CAP or FLP or any other, is a world in which you must assume imperfection from the components you use as building blocks. (Any theoretical "impossibility result" such as CAP or FLP is innately tied to its system model. This is the theoretical model of the world that the result depends on. Any such result doesn't really say that some goal—such as consensus—is impossible in general, but rather that it is impossible within that specific model. This can be extremely useful to practitioners in developing intuitions about which paths might be dead ends, but it can also be misleading if you only learn the result without learning the context to which the result applies.)

The real problem is that things break. The literature referred to here, such as FLP, is all about dealing with systems in which components are expected to fail. If this is the problem, then why don't we just use things that don't break, and then build better systems with components we can assume are robust?

Quite often in the past couple of years, the Spanner system from Google has been referenced as a justification for making this sort of assumption.[6] This system uses exactly the techniques mentioned earlier (NTP, GPS, and atomic clocks) to assist in coordinating the clocks of the hosts that make up Spanner and in minimizing and measuring (but not eliminating) the uncertainty about the differences between those clocks. The Spanner paper, along with the system it documents, is often used to back up claims that it is possible to have a distributed system with a single view of time.

Despite the appeal of pointing at Google and using such an argument from authority, everyone making that claim is wrong. In fact, anyone citing Spanner as evidence of synchronization being "solved" is either lying or has not actually read the paper. The simplest and clearest evidence against that claim is the Spanner paper itself. The TrueTime component of Spanner does not provide a simple and unified shared timeline. Instead, it very clearly provides an API that directly exposes uncertainty about differences in perceived time between system clocks. If you ask it for the current time, it does not give you a single value, but rather a pair of values describing a range of possibility around "now"—that is, TrueTime does the opposite of fixing this fundamental problem. It takes the brave and fascinating choice of confronting it directly and being explicit about uncertainty, instead of pretending that a single absolute value for "now" is meaningful across a working distributed system.

Within the production environment of Spanner, clock drift at any moment is typically from one to seven milliseconds. That is the best Google can do after including the corrective effects of GPS, atomic clocks, eviction of the worst-drifted clocks, and more in order to minimize skew. Typical x86 clocks vary their speeds depending on all kinds of unpredictable environmental factors, such as load, heat, and power. Even the difference between the top and bottom of the same rack can lead to a variance in skew. If the best that can be done in a wildly expensive environment like Google's is to live with an uncertainty of several milliseconds, most of us should assume that our own clocks are off by much more than that.

Another claim that is often made in justifying the "just pretend it's fine" approach in distributed systems design is that sufficiently high-quality equipment doesn't fail, or at least fails so rarely that you don't need to worry about it. This claim would be understandable, though incorrect, coming from the makers of such equipment, but it is usually the users of such gear, such as designers of distributed database software, who are heard making such claims. (The designers of GPS, which Spanner and others rely on, certainly didn't subscribe to this kind of claim. There are almost 30 GPS satellites, and you need to see only four of them for the system to work. Each of those satellites has multiple redundant atomic clocks so it can continue functioning when one of them breaks.)

One of the most common variants of such claims is "the network is reliable," in the context of a local, in-data-center network. Many systems have undefined and most likely disastrous behavior when networks behave poorly. The people who want to sell you these systems justify their choice to ignore reality by explaining that such failures are extremely uncommon. By doing this, they do their customers a double disservice. First, even if such events were rare, wouldn't you want to understand how your system would behave when they do occur, in order to prepare for the impact those events would have on your business? Second, and worse, the claim is simply a lie—so bald-faced a lie, in fact, that it is the very first of the classic eight fallacies of distributed computing.[7,15] The realities of such failures are well-documented in a previous ACM *Queue* article,[3] and the evidence is so very clear and present that anyone justifying software design choices by claiming that "the network is reliable" without irony should probably not be trusted to build any computing systems at all. While it is true that some systems and networks might not have failed in a way that a given observer could notice, it would be foolish to base a system's safety on the assumption that it will never fail.

The opposite approach to this waving-off of physical problems is to assume that almost nothing can be counted on, and to design using only formal models of a very adversarial world. The "asynchronous" model that FLP was proved on is not the most adversarial model on which to build a working system, but it is certainly a world much more hostile than most developers believe their systems are running in. The thinking goes that if the world you model in is worse than the world you run in, then things you can succeed at in the model should be possible in the real world of implementation. (Note, distributed systems theorists have some models that are harder to succeed in, such as those including the possibility of "Byzantine" failure, where parts of the system can fail in much worse ways than just crashing or delaying. For a truly adversarial network/system model, you could see, for example, either the symbolic or the computational models used by cryptographic protocol theorists. In that world, system builders really do assume that your own network is out to get you.)

It is in this context, assuming a world that is a bit worse than we think we are really operating in, that the best-known protocols for consensus and coordination, such as Paxos,[12] are designed. For such essential building blocks for distributed systems, it is useful to know that they can provide their most important guarantees even in a world that is working against the designer by arbitrarily losing messages, crashing hosts, and so on. (For example, with Paxos and related protocols, the most-emphasized property is "safety"—the guarantee that different participating hosts will never make conflicting decisions.) Another such area of work is logical time, manifest as vector clocks, version vectors, and other ways of abstracting over the ordering of events. This idea generally acknowledges the inability to assume synchronized clocks and builds notions of ordering for a world in which clocks are entirely unreliable.

Of course, you should always strive to make everything, including networks, as reliable as possible. Confusing that constant goal with an achievable perfect end state is silly. Equally silly would be a purist view that only the most well-founded and perfectly understood theoretical models are sensible starting places for building systems. Many of the most effective distributed systems are built on worthwhile elements that do not map perfectly to the most common models of distributed computing. An exemplary such building block would be TCP. This nearly ubiquitous protocol provides some very useful properties, the exact set of which do not map exactly to any of the typical network models used in developing theoretical results such as FLP. This creates a quandary for the systems builder: it would be foolish not to assume the reality of something like TCP when designing, but in some cases that puts them in a tenuous position if they try to understand how exactly distributed systems theory applies to their work.

The Zab protocol, which forms the most essential part of the popular Apache ZooKeeper coordination software, is a fascinating example of an attempt at walking this middle road.[10] The authors of ZooKeeper knew about existing consensus protocols such as Paxos but decided they wanted their own protocol with a few additional features such as the ability to be processing many requests at once instead of waiting for each proposal to complete before starting the next one. They realized that if they built on TCP, they had the advantage of an underlying system that provided some valuable properties that they could assume in their protocol. For example, within a single connected TCP socket, a sender producing message A followed by message B can safely assume that if the receiver reads B, then the receiver has previously read A. That "prefix" property is very useful, and is not present in the asynchronous model. This is a concrete example of the advantages available by looking at both the research in the field and the specific technology that is actually available to build on, instead of ignoring either.

When trying to be pragmatic, though, one must be careful not to let that pragmatism become its own strange kind of purity and an excuse for sloppy work. The Zab protocol as implemented inside ZooKeeper, the de facto reference implementation, has never been an accurate implementation of the Zab protocol as designed.[13] You might call yourself a "pragmatist" and note that most other software also doesn't match a formal specification; thus you might say that there is nothing unusual to worry about here. You would be wrong for two reasons. First, the role ZooKeeper and similar software is used for is not like other software; it is there precisely to provide essential safety properties forming a bedrock on which the rest of a system can make powerful assumptions. Second, if there are problems with the safety properties of a protocol like this, the appearance of those problems (while possibly very dangerous) can be subtle and easy to miss. Without a strong belief that the implementation reflects the protocol as analyzed, it is not reasonable for a user to trust a system. The claim that a system's "good-enough correctness" is proven by its popularity is nonsense if the casual user cannot evaluate that correctness.

All of this is not to pick on ZooKeeper. In fact, ZooKeeper is one of the highest-quality pieces of open-source software in its genre, with many excellent engineers continually improving it. Under recent analysis, it fares far better under stress than anything it competes with.[1] I have pointed at ZooKeeper only as an example of both the necessity and the pitfalls of taking a middle road with regard to theory and practicality. Mapping theory to practice can be extremely challenging.

Another example of this middle road is hybrid logical clocks[11] that integrate the general techniques of logical time with timestamps from physical clocks. This allows users to apply some

interesting techniques based on having a view (imperfect, but still useful) of the physical clocks across a whole system. Much like Spanner, this does not pretend to create a single unified timeline but does allow a system designer to build on the best available knowledge of time as perceived and communicated across a cluster.

All of these different coordination systems—including Paxos, Viewstamped Replication, Zab/Zookeeper, and Raft—provide ways of defining an ordering of events across a distributed system even though physical time cannot safely be used for that purpose. These protocols, though, absolutely can be used for that purpose: to provide a unified timeline across a system. You can think of coordination as providing a logical surrogate for "now." When used in that way, however, these protocols have a cost, resulting from something they all fundamentally have in common: constant communication. For example, if you coordinate an ordering for all of the things that happen in your distributed system, then at best you are able to provide a response latency no less than the round-trip time (two sequential message deliveries) inside that system.

Depending on the details of your coordination system, there may be similar bounds on throughput as well. The designer of a system with aggressive performance demands may wish to do things right but find the cost of constant coordination to be prohibitive. This is especially the case when hosts or networks are expected to fail often, as many coordination systems provide only "eventual liveness" and can make progress only during times of minimal trouble. Even in those rare times when everything is working perfectly, however, the communication cost of constant coordination might simply be too high.

Giving up strict coordination in exchange for performance is a well-known tradeoff in many areas of computing, including CPU architecture, multithreaded programming, and DBMS (database management system) design. Quite often this has turned into a game of finding out just how little coordination is really needed to provide unsurprising behavior to users. While designers of some concurrent-but-local systems have developed quite a collection of tricks for managing just enough coordination (for example, the RDBMS field has a long history of interesting performance hacks, often resulting in being far less ACID [atomicity, consistency, isolation, durability] than you might guess[2]), the exploration of such techniques for general distributed systems is just beginning.

This is an exciting time, as the subject of how to make these tradeoffs in distributed systems design is just now starting to be studied seriously. One place where this topic is getting the attention it deserves is in the BOOM (Berkeley Orders of Magnitude) team at the University of California Berkeley, where multiple researchers are taking different but related approaches to understanding how to make disciplined tradeoffs.[4] They are breaking new ground in knowing when and how you can safely decide not to care about "now" and thus not pay the costs of coordination. Work like this may soon lead to a greater understanding of exactly how little we really need synchronized time in order to do our work. If distributed systems can be built with less coordination while still providing all of the safety properties needed, they may be faster, more resilient, and more able to scale.

Another important area of research on avoiding the need for worrying about time or coordination involves CRDTs (conflict-free replicated data types). These are data structures whose updates never need to be ordered and so can be used safely without trying to tackle the problem of time. They provide a kind of safety that is sometimes called strong eventual consistency: all hosts in a system that have received the same set of updates, regardless of order, will have the same state. It has long been known that this can be achieved if all of the work you do has certain properties, such as being

commutative, associative, and idempotent. What makes CRDTs exciting is that the researchers in that field[16] are expanding our understanding of how expressive we can be within that limitation and how inexpensively we can do such work while providing a rich set of data types that work off the shelf for developers.

One way to tell that the development of these topics is just beginning is the existence of popular distributed systems that prefer ad-hoc hacks instead of the best-known choices for dealing with their problems of consistency, coordination, or consensus. One example of this is any distributed database with a "last write wins" policy for resolving conflicting writes. Since we already know that "last" by itself is a meaningless term for the same reason that "now" is not a simple value across the whole system, this is really a "many writes, chosen unpredictably, will be lost" policy—but that wouldn't sell as many databases, would it? Even if the state of the art is still rapidly improving, anyone should be able to do better than this.

Another example, just as terrible as the "most writes lose" database strategy, is the choice to solve cluster management via ad-hoc coordination code instead of using a formally founded and well-analyzed consensus protocol. If you really do need something other than one of the well-known consensus protocols to solve the same problem that they solve (hint: you don't), then at a very minimum you ought to do what the ZooKeeper/Zab implementers did and document your goals and assumptions clearly. That won't save you from disaster, but it will at least assist the archaeologists who come later to examine your remains.

This is a very exciting time to be a builder of distributed systems. Many changes are still to come. Some truths, however, are very likely to remain. The idea of "now" as a simple value that has meaning across a distance will always be problematic. We will continue to need both more research and more practical experience to improve our tools for living with that reality. We can do better. We can do it now.

ACKNOWLEDGMENTS

I would like to thank those who provided feedback, including Peter Bailis, Christopher Meiklejohn, Steve Vinoski, Kyle Kingsbury, and Terry Coatta.

REFERENCES

1. Aphyr. 2013. Call me maybe: ZooKeeper; http://aphyr.com/posts/291-call-me-maybe-zookeeper.
2. Bailis, P. 2013. When is "ACID" ACID? Rarely; http://www.bailis.org/blog/when-is-acid-acid-rarely/.
3. Bailis, P., Kingsbury, K. 2014. The network is reliable. ACM *Queue* 12(7); http://queue.acm.org/detail.cfm?id=2655736.
4. BOOM; http://boom.cs.berkeley.edu/papers.html.
5. Brewer, E. A. 2000. Towards robust distributed systems; http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.
6. Corbett, J. C., et al. 2012. Spanner: Google's globally distributed database. Published in *Proceedings of the 10th Symposium on Operating System Design and Implementation*; http://research.google.com/archive/spanner.html; http://static.googleusercontent.com/media/research.google.com/en/us/archive/spanner-osdi2012.pdf.
7. Deutsch, P. The eight fallacies of distributed computing; https://blogs.oracle.com/jag/resource/Fallacies.html.

8. Fischer, M. J., Lynch, N. A., Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2): 374-382; http://macs.citadel.edu/rudolphg/csci604/ImpossibilityofConsensus.pdf.
9. Gilbert, S., Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News* 33 (2): 51-59; http://webpages.cs.luc.edu/~pld/353/gilbert_lynch_brewer_proof.pdf.
10. Junqueira, F. P., Reed, B. C., Serafini, M. 2011. Zab: high-performance broadcast for primary backup systems; http://web.stanford.edu/class/cs347/reading/zab.pdf.
11. Kulkarni, S., Demirbas, M., Madeppa, D., Bharadwaj, A., Leone, M. 2014. Logical physical clocks and consistent snapshots in globally distributed databases; http://www.cse.buffalo.edu/tech-reports/2014-04.pdf.
12. Lamport, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16(2): 133-169; http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport-paxos.
13. Medeiros, A. 2012. ZooKeeper's atomic broadcast protocol: theory and practice; http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf.
14. NTP; http://www.ntp.org.
15. Rotem-Gal-Oz, A. Fallacies of distributed computing explained; http://www.rgoarchitects.com/Files/fallacies.pdf.
16. SyncFree; https://syncfree.lip6.fr/index.php/publications.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

JUSTIN SHEEHY is the site leader for VMware's Cambridge MA R&D center, and also plays a strategic role as architect for the company's Storage & Availability business. His three most recent positions before joining VMware were as CTO for Basho, Principal Scientist at MITRE, and Senior Architect at Akamai. Much of his career has focused on resilient distributed systems, both how to build them and how to apply them to solve business problems related to scalability and availability. Justin can be found on Twitter as @justinsheehy.