Standing on Giant Distributed Shoulders

**FARSIGHTED
PHYSICISTS OF
YORE WERE
DANGLED SMART!**

PAT HELLAND

If you squint hard enough, many of the challenges of distributed computing appear similar to the work done by the great physicists. Dang, those fellows were smart! Here, we examine some of the most important physics breakthroughs and draw some whimsical parallels to phenomena in the world of computing... just for fun.

NEWTON THOUGHT HE KNEW WHAT TIME IT WAS

Isaac Newton (1642 – 1727) was a brilliant physicist who defined the foundations for classical mechanics, laws of motion, and universal gravitation. He also built the first refracting telescope, developed a theory of color, and much more. He was one bad dude.

Newton saw time as constant and consistent across the universe. Furthermore, he assumed that gravity operated instantaneously without regard to distance. Each object in the universe is exerting gravitational force at all times.

This is very much like what we see in a single computer or in a tightly coupled cluster of computers that perform consistent work in a shared transaction. Transactions have a clearly defined local notion of time. Each transaction sees its work as crisply following a set of transactions. Time marches forward unperturbed by distance.

When I was studying computer science (and Nixon was

*It's Not Your Grandmother's
Database Anymore

president], we thought about only one computer. There was barely any network other than the one connecting terminals to the single computer. Sometimes, a tape would arrive from another computer and we had to figure out how to understand the data on it. We never thought much about time across computers. It would take a few years before we realized our perspective was too narrow.

EINSTEIN HAD MANY WATCHES

In 1905, Albert Einstein (1879 – 1955) proposed the special theory of relativity based on two principles. First, the laws of physics, including time, appear to be the same to all observers. Second, the speed of light is unchanging.

An implication of this theory is that there's no notion of simultaneity. The notion of simultaneity is relative to the observer, and the march of time is also relative to the observer. Each of these frames of reference is separated by the speed of light as interpreted relative to their speed in space.

This concept has some interesting consequences. The sun might have blown up five minutes ago, and the next three minutes will be lovely. When stuff happens far away, it takes time to find out... potentially a long time.

In computing, you can't know what's happening "over there." Interacting with another system always takes time. You can launch a message, but you always have to wait for the answer to come back to know the result. More and more, latency is becoming the major design point in systems.

The time horizon for knowledge propagation in a

The time horizon for knowledge propagation in a distributed system is unpredictable.

distributed system is unpredictable. This is even worse than in the physical Einstein-based universe. At least with our sun and the speed of light, we know that we can see what's happening at the sun as of eight minutes ago. In a distributed system, we have a statistical understanding of how our knowledge propagates, but we simply cannot know with certainty. The other server, in its very own time domain, may be incommunicado for a heck of a long time.

Furthermore, in any distributed interaction, a message may or may not be delivered within bounded time. Higher-level applications don't ever know if the protocol completed. Figure 1 shows how the last message delivery is not guaranteed and the sender never knows what the receiver knows. In any distributed protocol, the sender of the last message can't tell whether it arrived. That would require another message.

Another problem is that servers and messages live in their very own time spaces. Messages sent and received across multiple servers may have surprising reorderings. Each server and each message lives in its own time, and they may be relative to each other but may offer surprises because they are not coordinated. Some appear slower, and some faster. This is annoying.

In figure 2, as work flows across different times in servers and messages, the time is disconnected and may be slower or faster than expected. In this case, the second message sent by A may arrive after work caused by the first message, traveling through C. These problems can make your head hurt

1

FIGURE 1: **SENDER GETS NO CONFIRMATION OF FINAL MESSAGE DELIVERY**

in a similar fashion to how it hurts when contemplating twins where one travels close to the speed of light and appears to slow down while the other one stays home and ages.

You can't do distributed agreement in bounded time. Messages get lost. You can retry them and they'll probably get through. In a fixed period of time, however, there is a small (perhaps very small) chance they won't arrive. For any fixed period of time, there's a chance the partner server will be running *sloooooow* and not get back.

Two-phase commit cannot guarantee agreement in

2

FIGURE 2: **DISCONNECTED TIME MAY BE SLOWER OR FASTER THAN EXPECTED**

bounded time. Similarly, Paxos,⁷ Raft,⁸ and the other cool agreement protocols cannot guarantee agreement in a bounded time. These protocols are *very likely to reach agreement soon*, but there's no guarantee⁴. Each lives in its own relative world and doesn't know what's happening over there... at least not yet.

According to the CAP Theorem^{1,5} [standing for consistency, availability, partition tolerance], if you tolerate failures of computers and/or networks, you can have either classic database consistency or database availability. To avoid application challenges, most systems choose consistency over availability.

Two-phase commit is the anti-availability protocol.

From where I stand, Einstein made a lot of sense. I'm not sure how you feel about him.

HUBBLE WAS INCREASINGLY FAR OUT

Edwin Hubble (1889–1953) was an astronomer who discovered that the farther away an object is, the faster it is receding from us. This, in turn, implies the universe is expanding. Basically, everything is getting farther away from everything else.

In computing, we have seen an ever-increasing amount of computation, bandwidth, and memory. It looks like this will continue for a while. Latency is not decreasing too much and is limited by the speed of light. There are no obvious signs that the speed of light will stop being a constraint anytime soon. The number of instruction opportunities lost to waiting while something is fetched is increasing inexorably.

Computing is like Hubble's universe...

Everything is getting farther away from everything else.

Shared read-only data isn't the biggest problem. With enough cache, you can pull the stuff you need into the sharing system. Sharing writeable stuff is a disaster. You frequently stall while pulling a cache line with the latest copy from a cohort's cache. More and more instruction opportunities will be lost while waiting. *This will only get worse as time moves on!*

Shared memory works great...
as long as you don't *share* memory.

Either we figure out how to get around that pesky speed-of-light thing, or we're going to need to work harder on asynchrony and concurrency.

HEISENBERG WASN'T SURE

Werner Heisenberg (1901–1976) defined the uncertainty principle, which states that the more you know about the location of a particle, the less you know about its movement. Basically, you can't know everything about anything.

In a distributed system you have a gaggle of servers, each of which lives in various states of health, death, or garbage collection. The vast majority of the time you can chat with a server and get a crisp and timely result. Other times you don't get a prompt answer and it's tough to know if you should abandon the slacker or wait patiently. Furthermore, you don't know if the server got the request, did the work, and just hasn't answered. Any time a request goes to a single system, you don't know when the request will be delayed.^{2,6}

In some distributed systems, it is essential to have an extremely consistent and fast response time for online users. To accomplish this, multiple requests must be issued, and the completion of a subset of the requests is accepted as happiness.

In a distributed system, you can know *where*
the work is done or you can know *when*
the work is done but you can't know both.

To know *when* a request is done within a statistical SLA (service-level agreement), you need to accept that you don't know *where* the work will be done. Retries of the request are the only option to get a timely answer often enough. Hence, the requests had better be idempotent.

SCHRÖDINGER'S PUT

Erwin Schrödinger (1887–1961) was a leading physicist of the early 20th century. While he made many substantial contributions to quantum field theory, he is most often remembered for a thought experiment designed to show the challenges of quantum physics.

In quantum physics the theory, the math, and the experimental observations show that pretty much everything remains in multiple states until it interacts with or is observed by the external world. This is known as a superposition of states that collapse when you actually look.

To show that this seems goofy, Schrödinger proposed that this quantum-level uncertainty could map to a macro-level uncertainty. Start by placing a tiny bit of uranium, a Geiger counter, a vial of cyanide and a cat into a steel box. Rig the Geiger counter to use a hammer to break the vial of cyanide if an atom of uranium has decayed. Since the quantum physics of uranium decay show it is both decayed and not decayed until you observe the state, it is clear that the cat is simultaneously both dead and alive. Turns out many contemporary physicists think that it's not goofy... the cat would be in both states. Go figure!

New distributed systems such as Dynamo³ store their

data in unpredictable locations. This allows prompt and consistent latencies for **puts** as well as self-managing and self-balancing servers. Typically, the client issues a **put** to each of three servers, and when the cluster is automatically rebalancing, the destination servers may be sloshing data around. The set of servers used as destinations may be slippery. A subsequent **get** may need to try many servers to track down the new value. If a client dies during a **put**, it is possible that no servers received the new value or that only a single server received it. That single server may or may not die before sharing the news. That single server may die, not be around to answer a read, and then later pop back to life resurrecting the missing **put**.

Therefore, a subsequent **get** may find the **put**, or it may not. There is effectively no limit to the number of places it may be hiding. There's no upper bound on the time taken for the new value to appear. If it does appear, it will be re-replicated to make it stick.

While not yet observed,
a **put** does not really exist...
it's likely to exist but you can't be sure.
Only after it is seen by a **get** will the **put** really exist.

Furthermore, the failure to observe does not mean the **put** is really missing. It may be lurking in a dead or unresponsive machine. If you see the **put** and force its replication to multiple servers, it remains in existence with very high fidelity. Not seeing it tells you only that it's likely it's not there.

CONCLUSION

Wow! There have been lots of brilliant physicists, many of them not mentioned here. Much of their work has shown us the very counterintuitive ways the world works. Year after year, there are new understandings and many surprises.

In our nascent discipline of distributed systems, we would be wise to realize that there are subtleties, surprises, and bizarre uncertainties intrinsic in what we do. Understanding, bounding, and managing the tradeoffs inherent in these systems will be a source of great challenge for years to come. I think it's a lot of fun!

References

1. Brewer, E. A. 2000. Towards robust distributed systems. Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing.
2. Dean, J., Barroso, L. A. 2013. The tail at scale. *Communications of the ACM* 56(2): 74-80.
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. Proceedings of the 21st ACM Symposium on Operating Systems Principles: 205-220.
4. Fischer, M., Lynch, N., Paterson, M. 1985. The impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Vol 32, No. 2, April 1985.
5. Gilbert, S., Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, and partition-tolerant

web services. *ACM SIGACT News*, 33[2].

6. Helland, P. 2015. Heisenberg was on the write track. Seventh Biennial Conference on Innovative Data Systems Research.
7. Lamport, L. 1998. The part time parliament. *ACM Transactions on Computer Systems* 16[2] May 1998.
8. Ongaro, D., Ousterhout, J. 2014. In search of an understandable consensus algorithm. Proceedings of the Usenix Annual Technical Conference; <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Pat Helland has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.