Metadata

On distributed systems broadly defined and other curiosities. The opinions on this site are my own.

Hints for Distributed Systems Design

- October 02, 2023

This is with apologies to Butler Lampson, who published the "Hints for computer system design" paper 40 years ago in SOSP'83. I don't claim to match that work of course. I just thought I could draft this post to organize my thinking about designing distributed systems and get feedback from others.

I start with the same disclaimer Lampson gave. These hints are not novel, not foolproof recipes, not laws of design, not precisely formulated, and not always appropriate. They are just hints. They are context dependent, and some of them may be controversial.

That being said, I have seen these hints successfully applied in distributed systems design throughout my 25 years in the field, starting from the theory of distributed systems (98-01), immersing into the practice of wireless sensor networks (01-11), and working on cloud computing systems both in the academia and industry ever since. These heuristic principles have been applied knowingly or unknowingly and has proven useful. I didn't invent any of these hints. These are collective products of distributed systems researchers and practitioners over many decades. I have been lucky to observe and learn from several of them including Leslie Lamport, Nancy Lynch, Anish Arora, Pat Helland, and Marc Brooker and several other Amazon/AWS practitioners.

> *As to methods, there may be a million and then some, but principles are few. The man who grasps principles can successfully select his own methods. The man who tries methods, ignoring principles, is sure to have trouble. -- Harrington Emerson*

Organization

I organize the hints into three categories, functionality, performance, and fault-tolerance. Here they are altogether before we visit each one. For each hint, I give a primary example from state machine replication (SMR) and Paxos protocols field since I have been working on coordination in cloud systems. I also provide secondary examples from the more general distributed systems world.

Functionality:

- Apply abstraction
- Reduce coordination
- Embrace monotonicity

Performance:

- Prefer partial-order over total-order
- Leverage time
- Use indirection and proxies
- Simulate for estimation

Fault-tolerance/availability (does it keep working):

- Ingrain fault-tolerance
- Maintain a performance gradient
- Invest in deterministic simulation
- Feel the pain

Bonus hint: Design for the cloud

1. Apply abstraction

Slice out the protocol from the system, omit unnecessary details, simplify the complex system into a useful model, and then design your solution at that abstract plane. In that abstract plane, devoid of distractions, you have more freedom to explore the full range of the design space, and converge on a suitable solution. You then refine the solution back to the concrete plane with proper caution. My previous blog post was on this topic: "Beyond the Code: TLA+ and the Art of Abstraction". So I refer you there to keep this explanation short. The important point is to think abstractly and not to get bogged down in

the details. Don't miss the forest from the trees. Abstraction is a powerful tool for avoiding distraction, and coming up with innovative solutions at a higher plane.

> *The only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called "abstraction"; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worthwhile to point out that the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. --Edsger W. Dijkstra*

Let's give an example for this hint from the SMR/Paxos domain. The entire SMR/Paxos field is a testament to the power of abstraction. The SMR abstraction serves as a bedrock for building dependable distributed systems, including cloud systems at AWS, Meta, Google, Azure, and renowned databases like DynamoDB, Aurora, Spanner, MongoDB, and CockroachDB. SMR provides the developers with the abstraction of an infallible virtual mode which is in reality implemented over fallible physical notes by way of replicating operations that mutate/exercise the state machine as well as imposing a consistent ordering on these operations using Paxos family of consensus protocols. It's possible to refine and instantiate this abstraction in many different ways craft customized high-performance solutions for your system. For example chain-replicated SMR, primary-backup approaches, Raft replicated SMR over replicasets, and WPaxos replicated SMR over geo-distributed regions. (I plan to write a blog post later detailing these refinements.) Flexible quorums alone is a good example of the kind of jewels you can find hiding in the abstract plane for 30 years after invention of Paxos. When properly refined/instantiated, Paxos family of protocols can deliver top-notch performance and scalability matching those of unsafe eventual consistency protocols while providing fault-tolerance against all possible cornercases.

There are many other examples of this hint from the general distributed systems area. The simple map-reduce abstraction has jump-started the first generation of data intensive computing in the cloud, because it hid well the complexity of the underlying resource allocation, fault-tolerance, and data movement. Tensorflow

and other dataflow work generalized map-reduce.

2. Reduce coordination

Coordination introduces bottlenecks and increases the risk of failures and the blast-radius of failures. So it is best to avoid coordination in distributed systems as much as you can. Use coordination sparingly, primarily at the control plane, and minimize its presence in the data path. Even as a distributed coordination researcher and Paxos expert, my advice is to reduce coordination. Coordination is expensive, and if you can solve your problem without coordination, pursue that by all means.

To provide an example from the SMR systems, consider our work on WPaxos. WPaxos maps keys to different leaders to reduce coordination to isolated conflict (also independent failure) domains rather than using one conflict domain and imposing a total order on everything. Replicasets used in NoSQL and DistSQL systems are also examples of isolated conflict domains. Those are statically mapped, and you need to change the mapping out-of-protocol, wherease in WPaxos the mapping and key-movement across replicasets are embedded/integrated in to the protocol.

From the broader systems domain, a great example here is optimistic concurrence control (OCC). In OCC, each operation is executed without acquiring locks and conflicts are detected and result on when necessary. This is an example of where laziness pays well (of course for low conflict workloads). Another important realization of reducing coordination is separating the read path from the right path (as in LSMs), and being able to do reads without needing to coordinate with writes operating on the items.

3. Embrace monotonicity

Monotonicity ensures that as new information becomes available, the decisions and actions taken in the past are not invalidated. Monotonicity is a very effective way of reducing the coordination overhead. For performing coordination efficiently and safely in a distributed system, the insight is almost always to transform the problem in to a monotonic one. The nice thing with the monotonicity property is that is “pre-replicated”throughout the universe, thus

establishing consensus without communication. Ernie Cohen is a master of this. He quickly finds a way to transform any problem to one with a monotonic component, and pivots the protocol around this monotonic angle to produce an efficient and resilient design.

An example of monotonicity from SMR/Paxos world comes from the ballot numbers in Paxos. The ballot numbers are monotonically increasing and they preempt smaller ballot numbers. This is how within a single slot Paxos makes progress without jeopardizing the safety. The quorum vote collection, accompanies the preemptive monotonically increasing ballot number, to ensure that all the accepted values safe, and as needed re-propose them. This helps the algorithm to move forward without invalidating decisions taken in the past and building on them. The Vote and Paxos TLA+ specs I mention here explains this process in detail.

Here are other examples of embracing monotonicity from distributed systems at large. Using immutable data makes things trivially monotonic. Using idempotent operations is the same deal but in the operation space. LSMs are also a good example of embracing monotonicity. Use of multi-versions and timestamps are as well. Finally, CALM (consistency and logical monotonicity) line of work from Joe Hellerstein and collaborators are great examples of embracing monotonicity.

4. Prefer partial order over total order

We now move to the performance related tips.

Total order means every event is assigned a unique position relative to all others. Total order is expensive and hard to maintain in the presence of failures. Partial order means some events may not have a defined order relative to each other. Instead, events are ordered based on causality. There is no now in distributed systems. Partial order is flexible and effective in handling concurrency and failures. Partial order is useful for curbing tail latency. For example, by using quorum or erasure coding, you get best-K of N performance and eliminate tail latency. So prefer partial order over total order.

From the SMR/Paxos domain, WPaxos applies here again. Leaderless protocols from the EPaxos family apply as well. You don't need to serialize everything, you

can have partial order on things. From the deterministic database family, Detock paper is an example of using partial order.

A great example of this from the systems at large is using snapshot isolation (SI) semantics in transactions. SI is prevalent in industry because they don't curb performance (in the meanwhile the academia insists on publishing serializable transactions work). SI allows more partial order than serializable transactions can. This reduces tail latency in reads, as reads do not contend with writes or wait for them. You don't need to check for read/write conflicts, and you don't pay the extra cost for serialization. (Developers know how to deal with the write skew problem using "SELECT FOR UPDATE" when needed.)

5. Leverage time for performance

Don't depend on time for correctness, but use it to improve performance as needed. Use monotonic time, like hybrid logical clocks (HLCs), to implement a data maturation watermark to avoid the frothiness of now (Pat Helland coined this awesome expression). Fresh data/operations have potential volatility and instability. By implementing a data maturity watermark, we can make safe and quick decisions based on stable information. Note that this is in line with the embrace monotonicity hint.

In 2018, I had written a review of the area, clock synchronization, hardware, and software protocols for leveraging time for performance. Famously, Google Spanner introduced the use of atomic clocks and TrueTime API to achieve a strictly serializable distributedSQL system, that allow linearizable reads by just waiting the small clock epsilon to clear out the uncertainty.

In the absence of atomic clocks, hybrid logical clocks (HLC) is useful for performing post-hoc (coordination-free) consistent (as in snapshot consistency) reads of values across replicasets with some real-time affinity as well. This is the reason behind the popularity of HLCs in NoSQL and DistributedSQL databases.

A recent example of leveraging time comes from PolarDB. The system uses modification-time tracking table (MTT) to serve strongly-consistent reads. This protocol also makes use of embracing monotonicity (hint#3).

Some leaderless consensus protocols, such as Accord and Tempo, use clock synchronization to establish a timestamp-stability watermark to proxy for dependency tracking. Some leaderless consensus protocols also impose intentional delays on messages to achieve roughly in order delivery to multiple replicas to reduce conflicts in geo replication. (This is useful but you should also exercise caution not to fall in to the metastability trap (hint #9).

Examples from more the more general distributed systems domain goes back all the way to the "Practical use of synchronized clocks" paper, PODC 1991. Leases is an example of leveraging time to reduce coordination and exercise optionality.

6. Use indirection and proxies

Indirection and proxies is your first recourse in scaling distributed systems. "Any problem in computer science is solvable by adding another level of interaction." So when you are in a face with a bottleneck, try introducing interaction and proxies.

A good example of this from SMR space is our work on PigPaxos where we introduced leader proxies to resolve the leader I/O bottleneck in coordinating with followers. This is also an example of applying abstraction. Paxos doesn't prescribe a way for the leader to communicate with the followers. So nothing in safety of Paxos depends on whether this communication is done directly or via proxy-leaders, giving us the PigPaxos optimization.

Other examples include using load balancers, and cache, of course. When using caches, be wary of the metastability trap (see hint#9). Brooker has a great post on this and DynamoDB is designed to avoid this. While components such as caches can improve performance, DynamoDB does not allow them to hide the work that would be performed in their absence, ensuring that the system is always provisioned to handle the unexpected.

7. Simulate for estimation

Lightweight formal methods helps for designing correct and fault-tolerant protocols. But a correct protocol is useless if it doesn't have good performance as well. Marc Brooker explains this nicely in his "Formal method solves only half

of my problems" post. We need to think of performance and cost as we design the protocol. Simulation is a good way to get back-of-the-envelope estimations for performance (latency, throughput) and cost (capacity and monetary cost). Monetary cost is worth emphasizing here, as it is very important when designing real world systems.

You can write your simulations in Python or another programming language of your choice. The point is to have a prototype up. Like Brooks said, "Plan to throw one away. You will anyway." The simulation is a good way of failing fast. You can use simulation for identifying bottlenecks early on, then using the previously mentioned hints (like using indirection/proxies, partial order), you can see if you can alleviate the bottlenecks. Using simulation is also useful for planning for capacity and latency budgets and working backwards from them. With your simulation, you can also try your system with different workload patterns to see if this protocol is the right choice for that. If performance doesn't work, go back to the drawing board, to explore the design space further with abstraction and tools like TLA+. Nowadays formal methods also started to add support for some statistical checking support which I blogged about this earlier here. This mode helps for estimating which variant of protocol is best to implement.

An example of this for SMR domain comes from our paper on "Dissecting performance bottlenecks of strongly-consistent replication protocols". We used elementary queueing theory and simulations to quantify bottlenecks in Paxos based SMR systems. This helped us to see which Paxos variant is more suitable for a given deployment even before implementing and benchmarking it.

Other examples of this come from Marc Brooker. He is famous for coding up simulations of systems and exploring things there. Hats off to Brooker! If you are inside Amazon, search "code.amazon" for Brooker, and you will see many simulations he wrote to try things on his own. His blog also shows many applications of simulations for distributed systems issues.

8. Ingrain fault-tolerance

Ok, we are now in the fault-tolerance/availability category. I would also include security alongside fault tolerance/availability here, but I have little experience/work there, and can't cover it effectively.

Fault tolerance should be ingrained at the design phase, rather than come as an afterthought and added as a bolt-on. Retrofitting fault-tolerance is problematic. You miss opportunities and take some one way doors in protocol design. Retrofitting fault-tolerance afterwards can introduce performance/scalability problems, complexity, and cost. It can also introduce many corner cases for data consistency and durability. Bolted on fault tolerance may also make you prone to the metastability trap (see hint #9 again).

Examples from SMR/Paxos domain come from Paxos itself! The vanilla Paxos (or MultiPaxos for that matter) comes with batteries included. The fault tolerance is built into the normal execution of the protocol. The recovery code is constantly exercised. This results in predictability and less bugs/corner-cases in implementations/deployments. Compare this with leaderless consensus variations. The recovery is a separate protocol branch there, and as a result, there are nasty recovery related bugs there. Having a separate recovery mode also makes them prone to metastability failures as well. The system has to do more work when it is faced with faults. At the worse possible time, when the capability drops, we are forcing the system to do more work via recovery, that it wasn't doing before. This is the recipe for metastable failures.

AWS is big on static stability. Static stability is the characteristics for the data plan to remain at equilibrium in response to disturbances in the data plane. I this is without the use of a control plane at all, so that reckoning, so to speak, there is a nice Amazon builders library article which I mentioned here.

Other examples of building in fault-tolerance comes from stateless or soft-state services design. Those are nice when you can have them, because you can just restart those components or instantiate as needed without worrying about coordination. The paper on crash-only software is a great example.

9. Maintain a performance gradient

It is important to avoid metastability failures. I blogged about the metastability paper here, so I'll keep this short. Metastability means permanent overload with load throughput, even after the trigger is removed. Metastability is all about not DOSing yourself: avoid an asymmetric request response work. When can this happen? This happens if you overoptimize the common case. To prevent this, try

to avoid large modality in performance, and maintain a performance gradient (Aleksey's term) in your system.

DynamoDB provides a good example of how to maintain a performance gradient. As Marc Brooker writes "the killer feature of DynamoDB is its predictability." To repeat from hint#6: "While components such as caches can improve performance, DynamoDB does not allow them to hide the work that would be performed in their absence, ensuring that the system is always provisioned to handle the unexpected."

We can give SQL versus NoSQL as an example here. "The declarative nature of SQL is a major strength, but also a common source of operational problems. This is because SQL obscures one of the most important practical questions about running a program: how much work are we asking the computer to do?" This illustrates the tension between abstraction and performance. Abstractions (hint #1) should not hide important features like performance. On that topic, be careful about when that garbage collection may kick in.

10. Invest in deterministic simulation

Simulating your distributed system in a controlled deterministic environment can reveal potential correctness and recovery problems. A good example of this is Turmoil developed in AWS, which provides a Rust/Tokio based deterministic simulation environment. Turmoil helps the teams identify corner cases (including those involving failures and partitions), replay them deterministically, and see what's wrong quickly. Deterministic simulations also help with investigating code conformance to the protocol. Finally, deterministic simulation also helps for improving developer velocity with one box testing as developers can add features faster and with more confidence

FoundationDB provides a good use case for deterministic simulation testing: "Even before building the database itself, FDB team built a deterministic database simulation framework that can simulate a network of interacting processes using synthetic workloads and simulating disk/process/network failures and recoveries, all within a single physical process. FDB relies on this randomized, deterministic simulation framework for failure injecting and end-to-end testing of the correctness of its distributed database (with real code) in a single box."

And as another example, model based test case generation can identify when you have model your model based test case generation can identify coronary cases and vulnerabilities that may not be apparent during development models are important and it determines six simulations. You can run these corner cases and you can even do lineage driven testing on your implementation to check for code conformance.

11. Feel the pain

Observability is a key tool for developing highly scalable and highly available distributed systems. The system should prioritize feeling the pain, not masking it. Don't let the lack of pain fool you. Attribute the pain to the correct subsystem and react fast. There had been tracing solutions that had received a lot of interest. Observability (logging metrics, tracing and profiling) had been an important topic in dev-ops.

Tracing follows a request through the system, so in that sense, they are in-depth going vertical. In previous work, we had introduced a way to observe the system horizontally using consistent cuts obtained by hybrid logical clocks (HLCs), and showed the advantages of horizontal monitoring. HLCs enable you to take retrospective snapshot across replica sets and give you a consistent picture. This enables you to see performance as well as correctness and consistency anomalies. This is also useful for aligning the implementation with the high-level model of the system. This work did not get traction. I hope we will see more development and adoption in this direction going forward.

12. Bonus hint: design for the cloud

We are two decades into the cloud era, but distributed algorithms are still designed and evaluated with the dedicated physical box model from the 1970s. There the incentive is to use as much of the resources in the boxes as possible in equal portions so as not to be bottlenecked by one of the boxes, and so as not to leave unutilized wasted resource. Contrast that to the cloud, where the resource model is different: you pay for what you use. You are not bottleneck by the single box model because the servers are big and your node can be instantiated as multi-tenant with others. You can get your node can be scaled up

and down as needed. Thanks to disaggregation, bottlenecks are less of an issue. Instead of the physical box limited thinking, you should adopt the abundant thinking of the cloud. But we are not there yet; there aren't many examples of systems/protocols that make good use of the cloud model. This anomaly is cue (as per Kuhn) that there will be revolutionary science period where a new paradigm will emerge for designing for the cloud.

Aleksey, my former PhD student, gave a great example of cloud-based thinking in a recent work. This work showed that although EPaxos would seem to provide 2X throughput of Multi-Paxos in the physically limited box model, in the cloud model of pay-per-use, MultiPaxos provides 2X throughput of EPaxos. Give this blog post a read to follow this story.

Another example of rethinking SMR for the cloud era, this time from the cloud provider point of view, comes from Delos OSDI'20. Please read my blog post on this to learn more.

Post a Comment

Popular posts from this blog

The end of a myth: Distributed transactions can scale

- April 10, 2023

This paper appeared in VLDB'17. The paper presents NAM-DB, a scalable distributed database system that uses RDMA (mostly 1-way RDMA) and a novel timestamp oracle to supp ...

READ MORE >>

Foundational distributed systems papers

- February 27, 2021

I talked about the importance of reading foundational papers last week. To followup, here is my compilation of foundational papers in the distributed systems area. (I focused on the core distributed systems area, and c ...

READ MORE >>

Learning about distributed systems: where to start?

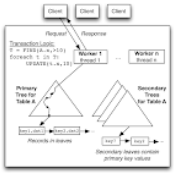
- *June 10, 2020*

This is definitely not a "learn distributed systems in 21 days" post. I recommend a principled, from the foundations-up, studying of distributed systems, which will take a good three months in the first pass, and many more months t ...

READ MORE >>

Speedy Transactions in Multicore In-Memory Databases

- *February 24, 2023*



This paper is from SOSP'13. The Polyjuice paper, which we studied last week, built on the Silo codebase and commit protocol, which led me to read the Silo paper. Silo is ...

READ MORE >>

Metastable failures in the wild

- *September 12, 2023*

This paper appeared in OSDI'22. There is a great summary of the paper by Aleksey (one of the authors and my former PhD student, go Aleksey!). There is also a great conferen ...

READ MORE >>

Amazon Aurora: Design Considerations + On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes

- *March 20, 2022*

Amazon Aurora is a high-throughput cloud-native relational database. I will summarize its design as covered by the Sigmod 17 and Sigmod 18 papers from the Aurora team. Aurora uses MySQL or PostgreSQL for the databas ...

READ MORE >>

SIGMOD panel: Future of Database System Architectures

- *July 05, 2023*

I mentioned this panel in my SIGMOD/PODS day 2 writeup. The panel consisted of (from right to left) Gustavo Alonso (ETH), Swami Sivasubramanian (AWS), Anastasia Ailamaki ...

READ MORE >>

The Seattle Report on Database Research (2022)

- *August 08, 2022*

Every 5 years, researchers from academia and industry gather to write a state-of-the-union (SOTU) report on database research. This one was released recently . It is a very readable report, and my summary consists of important p ...

READ MORE >>

There is plenty of room at the bottom

- *August 17, 2021*

This is a pun on the saying "there is always room at the top". This is also the title of a famous Feynman lecture from 1959 , where he made a case for nanotechnology. In this p ...

READ MORE >>

Murat Demirbas

MURAT

I am a principal applied scientist at AWS. On leave as a computer science and engineering professor at SUNY Buffalo. I work on distributed systems, distributed consensus, and cloud computing. You can follow me on Mastodon or Twitter.

VISIT PROFILE

Pageviews

 **3834408**

Recent Posts

Topics