Rethinking Datacenter Fabrics

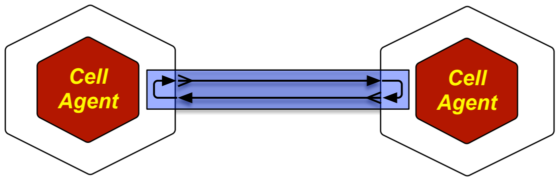
Many problems encountered in datacenters today arise from our inability to distinguish between a node that is merely slow from a node that has failed or become unreachable due to a network failure.

We take the two most recognizable elements in datacenters today: *servers* and *switches*, and refactor them into simpler, more foundational elements (fully *independent* failure domains): `cells` and `links`. A `cell` is a single type of node element (autonomous unit of compute, storage and packet processing). A `link` is an individual, bidirectional, computation object (an autonomous communication entity between *two* `cells`)[1].

A consequence of the former is that unifying node elements makes things simpler because we have only one type of node to manage instead of two. The consequence of the latter is profoundly more interesting: we raise the notion of a `link` to *first order* – a first-class citizen in the infrastructure – a bipartite[2] element of information with two *complementary* halves – *persistable* through failure and recovery events. i.e., a communication object that doesn't rule out that some fault-detection and computation is involved.

An example `link` utility is *The I Know That You Know That I Know* (`TIKTYKTIK`) property; which enables us to address some of the most difficult and pernicious problems in *distributed systems* today.

Another example `link` utility is *Atomic Information Transfer* (`AIT`). Unlike *replicated* state machines[3] used throughout distributed applications today, `links` are *single* state machines: the two halves of which maintain *temporal intimacy* through hidden packet exchanges. When a local agent or actor is ready, the `AIT` protocol transfers *indivisible* tokens across the link to the other agent, *atomically* (all or nothing)[4].



These `TIKTYKTIK` and `AIT` properties are *composable*. Trees of `links` provide a resilient *conserved quantities* mechanism to reliably distribute tokens among *agents* on an application graph. Intermediate cells *promise*[5] to never lose `AIT` tokens. This defends against lost tokens because if any part of the chain (or tree) breaks, alternate paths are available to seamlessly recover the conserved quanity and continue operation.

By strengthening the system model, `links` and `AIT` provide a general foundation to solve many distributed systems problems[6], such as failure-detection, consensus and distributed transactions.

Failure Modes

One might imagine we could achieve the properties of `links` over existing switched networks. If each host (or its NIC) maintains its half of the *shared state*, then shouldn't the switched network be able to act as a proxy for a single *logical* link? When a switched network fails, and reroutes, can't the two sides (NICs) just stitch the two halves of the *shared state* back together again?[7]

This simple hazard analysis[8] misses a fundamental issue: *networks don't maintain state on behalf of applications*. Switches drop packets (and state) whenever *they* feel like it, so there are many more ways for *logical links* to get confused over switched networks and compromise the integrity of the *shared state*.

Key issue: Switched networks may drop packets anywhere along the path; eradicating state *and* events needed to maintain *promises* and *liveness* respectively. When a `link` fails, both sides are preserved. If there is a failure in the *token transfer* it can always be detected, and retransmissions occur only on a *real* failure (such as disconnection–where alternative routes are explicitly coordinated with applications), thus enforcing that tokens have no duplicate or out of order deliveries on the `link`[9,10].

When packets are dropped in a switched network, more than information is lost, *events* are lost, and it becomes extraordinarily difficult to recover both sides of a *shared state* that stretches across even a single switch. A *directly connected* `link` can *promise* three things a switched network cannot[11]: (a) maintain an ordering of events (heal, send packets, fail, heal again, send more packets) – essential for non-idempotent and non-commutative operations. (b) not drop *certain* packets without notification – essential for recovery. And (c) maintain complementary *direction* state – essential for distributed self-stabilizing algorithms.

Bipartite Integrity

The *shared state* property is strengthened by mechanisms to recover from each type of failure. The more types of failures, the more complex and intractable this becomes. `Links` are independent failure domains, with (effectively) one failure hazard: *disconnection*[12]; which is straightforward to recover from. Switched networks, on the other hand, have many more failure hazards: they indiscriminately drop, delay, duplicate and reorder packets – that's just the way networks behave – justified by the end to end argument[13].

The shared state `TIKTYKTIK` property can also be used to mitigate broadcast storms in network rendezvous, timeout storms in microservices, or reconstruction storms in erasure coded storage[14]. In `AIT`, packets are not merely dropped, they are replaced with special events denoting failure, to maintain *liveness*. Because `link` failures are *independent* (unlike switched networks) we can successively recover individual disconnection failures. This paves the way for `AIT` to reverse *one or more* steps in distributed systems which use non-idempotent or non-commutative data structures.

Disconnection is the most likely failure hazard in `links`. Packets delayed by disconnected `links` don't threaten *liveness* or the integrity of the *shared state*. Switched network hazards include: indiscriminately *dropped*, *delayed*, *duplicated* and *reordered* packets. Conventional mitigations (e.g. TCP) add significant complexity and performance overheads, and still fail to solve the problem.

Examples

The advantage of the *shared state* is that both sides *know* the `link` is broken[15]. which can't be done through a switched network with *even a single switch* in series. `links` simplify some important distributed system algorithms such as two-phase commit, consensus and reliable tree generation:

Two-phase commit The prepare phase is asking if the receiving agent is ready to accept the token. This serves two purposes: communication liveness and agent readiness. `Links`[16] provide the communication liveness test, and we can avoid blocking on agent ready, by having the `link` store the token on the receiving half of the `link`. If there is a failure, both sides know; and both sides know what to do next.

Paxos "Agents may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted". The assumption is when a node has failed and restarted, it can't remember the state it needs to recover. With `AIT`, the other half of the link can tell it the state to recover from.

Reliable tree generation Binary link reversal algorithms[17] work by reversing the directions of some edges. Transforming an arbitrary directed acyclic input graph into an output graph with *at least one route from each node to a special destination node.* The resulting graph can thus be used to route messages in a loop-free manner[18]. `Links` store the direction of the arrow (head and tail); `AIT` facilitates the atomic swap of the arrow's tail and head to maintain loop-free routes during failure and recovery.

Conclusion: `Links` require a single physical connection; their benefits cannot be achieved over switched networks composed of a chain of *unreliable* links[19]. Combining the `TIKTYKTIK` and `AIT` properties with various classical algorithms, provides a general foundation to solve many distributed systems problems, as well as to mitigate broadcast, timeout and reconstruction storms in networks and distributed storage.

Notes

[1]Think of a `link` as a compute element with its own autonomous and independent failure domain; comprising the cable and NICs on both ends which form their own self contained execution environment. In the same way devices within a single execution environment are considered a single failure domain; the `link` can detect device and cable errors and stop the machine.

[2]This has nothing to do with biparitie graphs. Here, we mean the more classical definition of *bipartite* as "having two parts, or an agreement between two parties."

[3]Synchronization of timing domains in computers generally start from the processor clock on the motherboard, and fan out through the logic into the I/O subsystems. `AIT` is the link between two computers, and although it receives information from either side, it is not synchronized to either side. This "deep asynchronous domain" enables failure independence and atomicity.

[4]`links` are *exquisitely* sensitive to packet loss. This is intentional: we turn the FLP result *upside down*, and use a single failed process to guarantee the atomic property for `AIT`.

[5]The alternative definition of Futures/Promises also applies: execution entities primed to create future *events* for liveness.

[6]`links` also provide a more reliable foundation for distributed system *services* for applications: consensus, atomic broadcast, leader election, and distributed arithmetic, Which, in-turn, enable higher level functions such as *network-assisted-transactions* which serve, for example, distributable (scale-out), persistable (durable), and lightweight (but-accurate) accounting of *conserved quantities* associated with application state or resource usage.

[7]Such a recovery mechanism is *not* available through conventional switched networks; because of the uncertainty of how many packets were lost, exactly where along the path through the switched network they were lost, how many were duplicated, and how they might have been re-ordered in the switched network.

[8]Typical hazards: (1) Servers with a link to a single Top of Rack Switch (ToR) are unable to heal at all; there is only one path from the server to *anyone*. (2) ToRs represent SPoFs; when they fail, many servers (in that rack) also become unavailable. Worse still: the servers don't go down, they keep on computing but don't know they are isolated. ToRs have also been known to exhibit complex cascade failures where a firmware failure triggered in one will take down many neighbor TORs.

[9]Even without failures, the behavior of switched networks forces us into a high overhead approach. If packets can be indiscriminately dropped, delayed, duplicated and reordred, we have no choice but to implement `TCP/IP`, or something very much like it (which is unlikely to have the maturity and robustness of the existing tried and tested implementations). However, even in optimal situations, TCP introduces high overhead for *all* operations, which is too slow (e.g. Coherent Distributed Shared Memory Systems). There is no escape: if you have dropped, delayed duplicated or reordered packets, you have to effectively implement TCP. If we can eliminate duplication and re-ordering, recovery is far simpler, and removes the overhead from normal operations.

[10]Links do not reorder or duplicate packets so we can now use a high-performance `udp`-like protocol for coherence; only paying the performance cost of `TCP` when it fails for real, i.e. disconnection. Remarkably, this also paves the way for very high bandwidth utilization for datacenter to datacenter application flows, because it eliminates the most difficult aspects of reordering and duplication of packets

[11]Actually, a `link` can promise many more than just these three things: whatever property the agent, actor or application wishes to attach to the bipartite `link` object.

[12]In any physical system it is possible to drop packets, it will be much rarer but it is still possible. `links` can recover from individually dropped or corrupted packets, and *shared state integrity* can be maintained.

[13]The end-to-end principle states that in a general-purpose network, application-specific functions ought to reside in the end hosts of a network rather than in intermediary nodes, provided that they can be implemented "completely and correctly" in the end hosts. Our claim is that it is impossible to implement *bidirectional* synchronization primitives "completely and correctly" without `AIT` (or something very much like it). The datacenter is not the Internet, and Saltzer, Reed, & Clark considered only a careful (unidirectional) file transfer, not the *bidirectional* synchronization of replicas (coherency).

[14]`Links` also coalesce heartbeats to make microservices more scalable, and make failure detectors *reliable*.

[15]The *shared state* can be compromised by duplicated or reordered packets, but it is resilient to lost or delayed packets.

[16]`Links` exploit a unique combination of physics, electrical engineering and computer science. Think of `AIT` as the distributed systems (message-passing based) equivalent of the atomic Compare And Swap (`CAS`) primitive used in shared memory architectures. The result is handshake-free messaging with strong liveness, recoverability and security properties.

[17]Charron-Bost et. al. generalize the Gafni-Bertsakas (GB) binary link reversal algorithm.

[18]Link reversal algorithms don't generate shortest paths, just some paths. However, they do generate *multiple* loop-free routes. This allows the link to inform the agent (and the application, if needed) when switching to an alternate failover tree. As long as precomputed failover paths are available, we used the Dynamic Tree Algorithm (DTA). Only when cells lose *all* their paths do they need to participate in the link-reversal algorithms, which means less communication overhead and more stability.

[19]The shared state integrity of the single physical link is a *promise* that two NIC's can make with each other only over a single physical link. This is explicitly an *anti-promise* for conventional switched networks.