A Design for Resilient Datacenter Networks

Alan H. Karp and Paul L. Borrill
EARTH Computing

***Abstract*—The datacenter networks we use today are based on a 50-year-old design. While it has served us well, that design no longer meets the needs of today's distributed applications. The result is failures visible to the users of those applications in spite of developers' best efforts. We believe it is time to provide a new design, one that takes into account what today's applications need. To that end, we have developed a network architecture that starts at the physical layer and extends to the way applications communicate and how they are managed. A key design goal is the ability to recover from network failures so fast that applications perceive an unbreakable network.**

Datacenter networks were designed at a time when the primary use case was block transfer of files. People learned that they could meet their design goals if the switches connecting the servers were allowed to drop, reorder, delay, and duplicate packets. Any issues that arose from those decisions were not the network designers' problem.

The primary use case today is microservices sending short, latency sensitive messages to each other. Delayed or dropped packets can trigger timeouts, leading the application to falsely assume that the network has partitioned. The result is consistency failures or entering a recovery mode that can interfere with other communications, leading to even more timeouts. Such failures need to be handled in the application, which increases complexity for developers and reduces business agility.

Our design, which we call Ariadne, starts with the physical topology of the network and extends all the way to application management. A key feature is a network that heals from failures so fast that the applications do not even know they occurred.

Digital Object Identifier 10.1109/MITP.2020.3013511
Date of current version 6 November 2020.

PHYSICAL

Ariadne directly connects servers having standard Ethernet smart NICs with at least six ports and a general purpose CPU. The number of ports is close to the sweet spot for our algorithms. Fewer than six, and there is not enough path diversity to avoid partitions, while more than 10 slows down recovery from

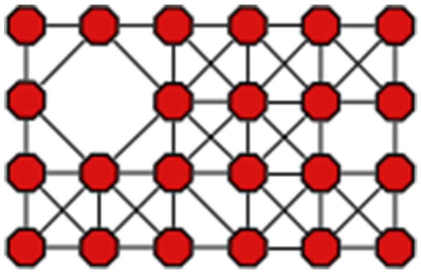

Figure 1. Physical topology for the Ariadne datacenter fabric. It is basically a mesh, but it need not be regular. The design adapts to miswired cables as well as missing nodes and links.

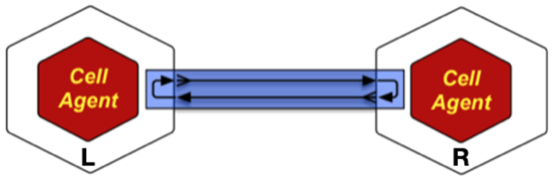

Figure 2. Link connecting two cells. The blue bar indicates that the link is a computational object capable of implementing Ariadne's link protools. The Cell Agent on each cell provides higher level functions, such as interacting with the host operating system.

a failed node. Ariadne does not need the resulting mesh to be perfect, as illustrated in Figure 1. It tolerates missing servers and links in addition to mis-wired cables.

This topology has some advantages. Neighbors communicate directly instead of through a top-of-rack switch, resulting in orders of magnitude lower link latency. Multipathing provides very high aggregate bandwidth between any pair of nodes. Many packets sends can be done in parallel instead of being serialized over one or two ports.

LINK PROTOCOLS

Above the physical layer is our software that controls the links. The novel idea is to treat the link as a computational object. On our fabric that is a 6-inch piece of copper wire, so we use a little bit of the NIC on each side, denoted by the blue bar in Figure 2. The computation the link can do is limited, but it is enough to support the two Ariadne link protocols.

Figure 2 shows two *cells*, each combining compute, storage, and packet processing. The Cell Agents provide higher level functions, such as setting up the information needed to forward packets.

Liveness

The liveness protocol replaces heartbeats and timeouts with events. The left cell (L) sends a packet to the right cell (R), which sends a packet to L. L and R take turns, so we call this a *ping pong* protocol. If L has no data to send, it sends a liveness packet to R. If L has data to send, it sends that packet. Even if L has more than one data packet to send, it still waits for a return packet from R before sending another one.

Waiting in this way has a number of advantages. Waiting is hop by hop credit-based flow control. There is no guessing about buffer sizes. Each port needs an outgoing slot for each of the other ports and one for applications running on that cell. Recovery from a broken link is simple since the packet count on the two sides never differs by more than one.

Most importantly, a lost packet stops forward progress on that link. Stopping means the two sides do not get into an inconsistent state because of a mistaken assumption about the other side, as often happens with timeouts.

Common Knowledge

Common knowledge[1] (CK) is the key to simplifying many distributed algorithms. However, it is impossible to achieve on asynchronous networks like the ones we use today.[2] The ping pong protocol violates one of the assumptions behind the impossibility proofs,[3] which means you can do some things on Ariadne that are impossible to do on today's networks.

At the end of two CK round trips, both sides know where the data is, and each side knows that the other side knows. The protocol can recover from a failure at any step of the process.

CK can be used in many places where people use consensus or transactions today, but CK is

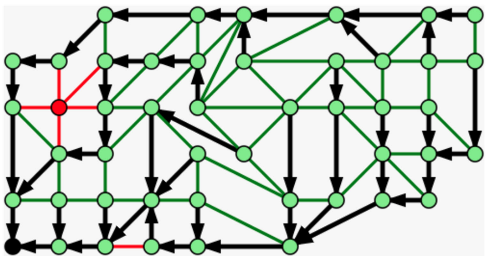

Figure 3. Handling link and cell failures. The root of the tree shown is in the lower left corner as a black circle. Red denotes a failed component. Note that the tree is intact.

much lighter weight. In another use case, a database 2-phase commit (2 PC)[2] has a rare failure mode that does not happen with 3 PC,[4] but people still risk using 2 PC rather than paying for an extra round trip. Since the middle round trip in 3 PC is independent of application state, it can be done with CK between the NICs. The result is the safety of 3 PC with the performance of 2 PC.

What if the receiving side does not want the data? Maybe its host OS is rebooting. Normally, it would send a negative acknowledgment, and the two sides would get their states back in sync. Instead, Ariadne's CK protocol runs the state machines backward to the initial state. Now, both sides are automatically in sync.

If you squint and look at it sideways, it is almost as if you had run time backwards. That means the next try is effectively the first time. The result is that the application sees exactly once delivery instead of the at most once or at least once that developers settle for today.

ADDRESSING

Address management is one of the most complex components of any microservices infrastructure.[5] Every microservice needs to know the address of every other microservice it sends messages to, even when the target moves. That requirement complicates migration, elasticity, and failure recovery. Unfortunately, using domain names with DNS does not always work well. Address lookup adds too much latency, and the environment is so dynamic that address caching is often ineffective.

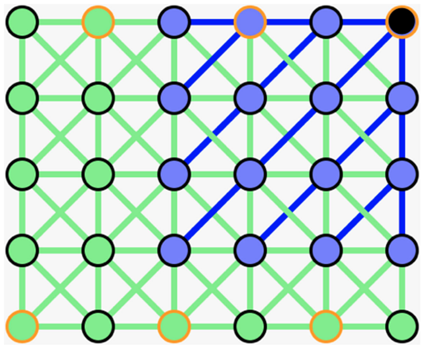

Figure 4. Subsetting a larger tree to produce the blue tree. The black cell is the root of the tree. Orange circles denote cells that talk to the outside world. This tree was built by sending an equation $hops < 4$ on the larger tree.

Each Ariadne cell builds a spanning tree rooted on itself when it is first connected. Figure 3 illustrates such a tree with its root in the lower left corner. For each tree, each cell records which port points to the parent on the tree, which ports point to the cell's children on that tree, and which ports are connected to links not on that tree, which we call *pruned* links.

When the root sends a message leafward, it reaches all cells on the tree; the root does not need to know exactly which cells will receive the message or even how many will. When a cell sends a message rootward, it reaches the root without the sending cell needing any additional information. Because a tree has exactly one path between any pair of nodes, it is easy for Ariadne to deliver packets in order.

Most of the time, you do not want to talk to the entire datacenter. Ariadne's tree-based addressing has a mechanism that allows an application to create a subset of a tree. You send a *Graph Virtual Machine* (GVM) equation on a tree, which cells on that tree evaluate in their local context. A cell joins the subtree if the result is TRUE. The equation $hops < 4$ was used to produce the blue tree in Figure 4.

FAILURE HANDLING

Consider the red link in the bottom row of Figure 3. It was the parent link for the cell to its right, the color indicating that the link failed. When the link connected to its parent fails, the cell does a guided depth first search to find a new path. The algorithm first tries the pruned link having the smallest hop count to the cell on the other side of the broken link. If there are no pruned links, the cell tries an arbitrary child.

A lost packet is treated as a link failure. If events are being received on some links but not others, the cell declares the nonresponsive links failed. If such a link starts responding, it is treated as being a pruned link that is available for failover.

The failover procedure typically takes only a handful of packet transfers, which we estimate will take a few microseconds. Also, since there is only one packet that might have been lost when the link failed, sewing up the in-order packet flow is trivial.

One of the hardest problems in distributed systems is failure detection when nodes can become temporarily nonresponsive. Timeouts can lead other nodes to conclude that a nonresponsive node is dead when it is merely slow, causing problems when it starts communicating again. One example is the double leader problem in consensus protocols, such as Paxos.[6] This behavior is key to the proof that a single failure can prevent reaching consensus on an asynchronous network.[3]

Ariadne's failover algorithm avoids this problem. If a cell fails, each neighbor sees a broken link. The failover algorithm will not find a path to the failed cell, so all neighbors will know that the cell has failed. We can easily restart a microservice that was running on the dead cell. As long as the new instance is put on the same tree as the original, everything just works.

This algorithm works even if the unreachable cell is temporarily nonresponsive. Once a failover message from one neighbor of the suspect cell reaches another neighbor of that cell, the receiving cell will declare its link to the suspect neighbor broken if there is no pong to its ping. Even if the suspect neighbor subsequently tries to communicate, the neighbor will not ping its pong. If no path is found, the suspect cell will not be allowed to rejoin the network until it has made its state consistent with that of its neighbors.

ORCHESTRATION

One of the messages we can send on a tree is a description of the microservices to deploy. Compare that procedure to what it takes to deploy a microservices application today. You must enumerate the servers, their addresses, and the addresses of the microservices they need to talk to, usually in a configuration file.[5]

With the GVM you simply create a subset of a tree and deploy the application on that subset tree. You do not need to know the exact set of cells or even how many are in the set. Task migration is far simpler since a task's address does not change if it stays on the tree when it moves. Load balancing, failover, and elasticity decisions can be made locally, as opposed to being made by the central traffic managers in service meshes[7] that still have to deal with the vagaries of the network.

Say that blue tree in Figure 4 shows the largest desired configuration for a sharded data store. At first, the direct neighbors of the root can handle all the data. When the cell to the left of the root in the picture gets overloaded, it can start forwarding requests to one or more of its children. No other cells need be involved. With this procedure, we should be able to make elasticity decisions in milliseconds instead of the seconds it takes today.

SECURITY

Vulnerabilities abound, and breaches occur on a regular bases. Typically, a successful breach of an application running in a datacenter is leveraged into an escalation of privilege attack against the operating system. The attacker then sends malformed packets on the network. Watching the resulting traffic gives the attacker the means to mount attacks on other servers. Essentially, every server in the datacenter is at risk if even one application has an exploited vulnerability.[8] Ariadne makes entire classes of these attacks inexpressible.

In Ariadne an application may only name a tree if explicit permission was granted. Even the operating system running on a server may only name the union of trees of the applications it runs. The result is confinement; the attacker can only reach a subset of the datacenter. Proper use of the GVM can limit which cells are at risk. For example, each cell can be assigned a sensitivity level, and the GVM equation for an untrusted application can tell only low sensitivity cells to join the tree.

Ariadne implements tenant management in a novel way. Each tenant is assigned a contiguous set of cells. The Cell Agent on each cell maintains a mask that effectively turns off any ports connected to other tenants. As far as the tenant is concerned, the corresponding links do not exist. The result is that one tenant is not vulnerable even if another tenant is breached. An additional benefit is that a tenant can create subtenants without needing to interact with the datacenter manager.

A successful attack against the trusted computing base (TCB) of a system allows the attacker to bypass all security mechanisms. Ariadne's TCB is protected by running it on a unikernel running on the processor on the NIC. Hardware and side channel attacks mounted from applications or the host operating system do not reach that CPU.

CONCLUSIONS

Over the years, several projects have tried to address the issues caused by decisions made long ago, but few have dared to touch the basic structure of the network or the protocols used on it. As a result, those efforts were only partially successful. Ariadne is fundamentally different, starting with a different topology, using a new set of link protocols, and providing an addressing scheme better suited to today's applications.

A key design goal of Ariadne is to fix in the network problems that occur in the network. Ariadne can do that so quickly that applications, even legacy applications tunneled through Ariadne, see a network that never appears to break. Also, you do not need to build a new datacenter. An Ariadne rack or two in the middle of an existing datacenter can run some key applications.

Ariadne is mostly scale invariant. Each cell only knows about itself and its immediate neighbors. In addition, each cell needs only about 100 B for every tree passing through it, which means even having millions of trees is manageable. The result is that Ariadne is suitable for deployments ranging from a single rack to an entire hyperscale datacenter.

We believe that many uses of heavyweight protocols, such as consensus and two-phase commit, can be replaced by CK across a link. Applications that take advantage of that feature will be qualitatively better for it.

At the time this article was written, Ariadne existed only in a simulator and a prototype implementation of the link protocols in a Linux driver. That is enough to allow us to validate the design's addressing and resilience properties and experiment with toy applications.

ACKNOWLEDGMENTS

Invaluable feedback was provided on this design by C. Baquero, P. Almeida, J. Vicary, and M. Burgess, as well as members of our team, B. Jackson, S. Chalmers, and B. Borden. Douglas Crockford's careful reading was a significant help.

REFERENCES

1. J. Y. Halpern and Y. Moses, "Knowledge and common knowledge in a distributed environment," *J. ACM*, vol. 37, no. 3, pp. 549–587, Jul. 1990.
2. J. Gray, "Notes on database operating systems," IBM Res. Lab., San Jose, CA, USA, Aug. 1987, p. 73. [Online]. Available: https://cs.nyu.edu/courses/fall18/CSCI-GA.3033-002/papers/Gray1978.pdf
3. M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.
4. D. Skeen, "A quorum-based commit protocol," Tech. Rep. TR82-483, Cornell Univ., Ithaca, NY, USA, 1982, [Online]. Available: https://ecommons.cornell.edu/handle/1813/6323
5. Kubernetes. 2020. Online. Available: https://kubernetes.io/docs/concepts/
6. L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.
7. Microsoft. 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-monitoring
8. I. Ghafir and V. Prenosil, "Advanced persistent threat attack detection: An overview," *Int. J. Adv. Comput. Netw. Secur.*, vol. 4, no. 4, pp. 50–54, 2014.

Alan H. Karp is currently a Principled Architect with EARTH Computing, Palo Alto, CA, USA. He received the Ph.D. degree in astronomy in 1974 from the University of Maryland, College Park, MD, USA. He is the recipient of two IBM Outstanding Innovation Awards and is coauthor on two Best Papers. He has authored or coauthored nearly 100 refereed papers and conference proceedings and holds more than 75 patents. He is a Senior Member of the IEEE and ACM and is a Member of the IEEE Computer Society. Contact him at alan@earthcomputing.io.

Paul L. Borrill is the founder and CEO of EARTH Computing in Palo Alto, CA, USA. He received a Ph.D. degree in physics in 1985 from University College London, London, U.K. After a stint at NASA, he held positions at Sun Microsystems, Quantum, Veritas Software, and Apple. He is the Founding Chairman of the Storage Networking Industry Association and served on the Board of Governors of the IEEE Computer Society as VP of Technical Activities and VP of Standards. EARTH Computing is Paul's second startup. Contact him at paul@earthcomputing.io.