Ariadne: A Design for Resilient Datacenter Networks[*]

Alan H. Karp and Paul L. Borrill
EARTH Computing
{alan, paul}@earthcomputing.io

EARTH Computing Technical Report 2020-1

June 3, 2020

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.

1 Introduction

Datacenter networks were designed at a time when the primary use case was block transfer of files, and the big problem was making effective use of the available bandwidth. Back then, most servers had one, or at most, two network ports, which required switches to connect large numbers of them. People learned that they could meet their design goals if the switches were allowed to drop, reorder, delay, and duplicate packets. Any issues that arose from those decisions weren't the network designers' problem. TCP was implemented to shield developers from many of those issues.

The world has changed. 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

[*]This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

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 the developer and reduces business agility.

Over the years, several projects have tried to address the issues caused by those decisions made so 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.

It often feels that our datacenter network has us trapped in a maze of complexity. That's why we call our design Ariadne in honor of the princess who gave Theseus a ball of string so he could find his way out of the labyrinth after slaying the Minotaur.

Ariadne starts from the ground up, with the physical topology of the network. That topology allows us to use link protocols specifically designed for distributed applications even though we use off the shelf smart network interface cards (NICs). Those protocols, in turn, enable us to use a unique form of addressing, one that frees the distributed application runtime from many of the issues it must deal with today. The result is a network architecture that dramatically simplifies orchestration of elastic services while healing from failures so fast that the applications don't even know they occurred.

In this paper, we describe the design of Ariadne starting with the physical layer (Section 2), then explaining the link protocols (Section 3), going on to describe our addressing scheme (Section 4), how we deal with failures in Section 5, and finishing by explaining how this design simplifies orchestration (Section 6). At each step, we'll focus on how the design leads to a resilient network. The Section 8 describes how others address problems caused by the network. We describe the current status of our work in Section 9 and finish with some conclusions in Section 10.

2 Physical

There are smart NICs on the market with a general purpose CPU having gigabytes of RAM and at least six ports [1]. That's enough ports to connect a large number of servers without switches, so that's what we do. As you can see in **Figure 1**, Ariadne doesn't need the mesh to be perfect. She tolerates missing servers and links in addition to mis-wired cables.

Ariadne has a number of advantages. The number of ports is close to the sweet spot for our algorithms. Fewer than six, and there isn't enough path diversity to avoid partitions, while more than 10 slows down recovery from a failed node. Multi-pathing provides very high aggregate bandwidth between any pair of nodes. Neighbors communicate directly instead of through a top-of-rack switch, resulting in orders of magnitude lower link latency.

There is no need to replace existing infrastructure to take advantage of this topology. Just add an 8-port smart NIC to an existing server that already has two ports on its motherboard. Legacy applications can use the motherboard

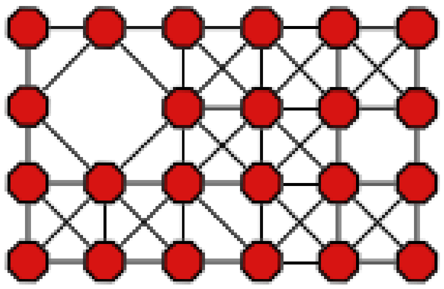

Figure 1: The physical topology for the Ariadne datacenter fabric. It is basically a mesh, but it need not be regular. The design tolerates mis-wired cables as well as missing nodes and links.

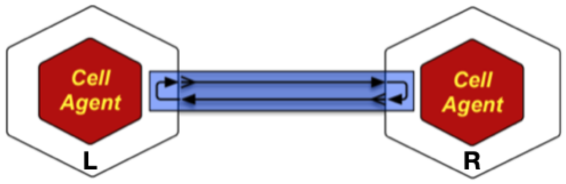

Figure 2: A link connecting two cells. The blue bar indicates that the link is a computational object capable of implementing Ariadne's link protoools. The Cell Agent on each cell provides higher level functions, such as interacting with the host operating system.

ports; applications that wish to take advantage of features of Ariadne use the smart NIC ports.

A mesh network, even an irregular one, isn't novel. It's how our design uses it that is, as the following sections show.

3 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's a 6-inch piece of copper wire, so we use a little bit of the NIC on each side, indicated by the blue bar in **Figure 2**. The computation the link can do is limited, but it's enough to support the two Ariadne link protocols.

Figure 2 shows two *cells*. Each cell combines compute, storage and packet processing. The Cell Agent shown on each side of the link provides higher level functions, such as connecting to the host operating system and setting up the information needed to forward packets.

A standard network has one packet protocol on the link and uses higher level functions to handle such things as liveness and consensus. For example, heartbeats with timeouts are typically used to detect liveness. We have chosen to use two protocols at the link level.

Liveness

The liveness protocol replaces heartbeats and timeouts with an event driven model. 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.

That sounds crazy. L has more data to send but has to wait? What a waste of bandwidth! The good news is that waiting provides a number of benefits.

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

Most importantly, a lost packet stops forward progress on that link. That's a good thing. It 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 [2] (CK) is the key to simplifying many distributed algorithms. However, it is impossible to achieve on asynchronous networks like the ones we use today [3]. The ping pong protocol violates one of the assumptions behind the impossibility proofs [4], 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 tolerates a failure at any step of the process. Should the link break, the two sides can sew up the packet stream when they find a new path between them as described in Section 5. Should a packet get lost, the two sides of the link freeze. When the cells detect that freeze, they revert to the failed link procedure. Should one of the servers fail, the survivor knows what to do based on what step the failure occurred.

Common knowledge can be used in many places where people use consensus or transactions today, but it's much lighter weight. In another use case, a database 2-phase commit (2PC) [3] has a rare failure mode that doesn't happen with 3PC [5], but people still risk using 2PC rather than paying for an extra round trip. It turns out that the middle round trip in 3PC is independent of application state, which means it can be done with common knowledge between the NICs. The result is the safety of 3PC with the performance of 2PC.

What if the receiving side doesn't want the token? 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's almost as if you had run time backwards. That means the next try is the first time. The result is the holy grail for distributed computation, exactly once delivery [6] instead of the at most once or at least once that developers settle for today.

4 Addressing

Address management is one of the most complex components of any microservices infrastructure. Every microservice needs to know the IP address of every other microservice it sends messages to, even when the target moves [7]. That requirement complicates migration, elasticity, and failure recovery. Unfortunately, using domain names with DNS doesn't work well. Address lookup adds too much latency, and the environment is so dynamic that address caching is often ineffective.

In a CLOS network, you build a spanning tree on the switches so there is a path between any pair of servers. Ariadne does something similar; each cell builds a spanning tree rooted on itself. **Figure 3** illustrates such a tree with its root in the lower left corner.

That sounds like a lot of data to maintain, N trees for N cells. However, that data structure never exists anywhere. Each cell has $O(1)$ data for each tree, basically, the port pointing to the parent on the tree, the ports pointing to the cell's children on that tree, and which ports are connected to links not on that tree, which we call *pruned* ports.

When the root sends a message leafward, it reaches all cells on the tree; the root doesn't 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.

Of course, most of the time, you don't 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. It sends out a Graph Virtual Machine (GVM) equation on a tree. Cells on that tree evaluate the equation in their local context. Only if the result is TRUE does the cell join the subset tree. The equation $hops < 3$ was used to produce the blue tree in **Figure 4**.

Section 6 shows the benefits of the GVM for elasticity, and Section 7 explains how it provides strong security guarantees.

5 Failure Handling

Consider the red link in the bottom row of Figure 3. It was the parent link for the cell to its right, but the red color indicates that the link failed. When the

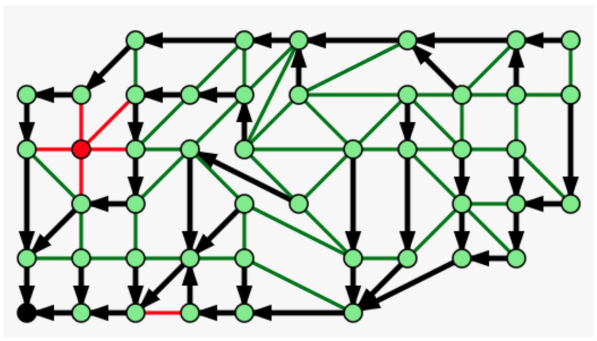

Figure 3: Handling link and cell failures. The root of the tree shown is in the lower left corner as a black circle. Red denotes a failed component. Note that the tree is intact.

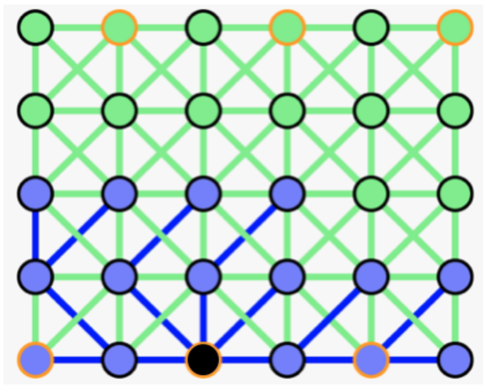

Figure 4: Subsetting a larger tree to produce the blue tree. The black circled is the root of the tree. Orange circles denote cells that talk to the outside world. This tree was built by sending an equation $hops < 3$ on the larger tree.

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. If there are no pruned links, the cell tries an arbitrary child.

A lost packet, which is extremely rare in point to point communication, is treated as a link failure. If events are being received on some links but not others, the cell declares the non-responsive links to be broken. If such a link starts responding, it is treated as being a pruned link on all trees.

In the common case, the failover procedure takes only a handful of packet round trips, which we estimate will take a few μs. 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 non-responsive. Timeouts can lead other nodes to conclude that a non-responsive node is dead when it is merely slow, causing problems when it starts communicating again. The effect is seen in the double leader problem in consensus protocols, such as Paxos [8]. This behavior is key to the proof that a single failure can prevent reaching consensus [4].

Ariadne's failover algorithm avoids this problem. If a node fails, each neighbor sees a broken link. The failover algorithm won't 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 cell is merely non-responsive rather than dead. 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 won't ping its pong. If no path is found, the suspect cell won't be allowed to rejoin the network until it has made its state consistent with that of its neighbors.

6 Orchestration

One of the messages we can send on a tree is a manifest describing the microservices to deploy. Compare that procedure to what it takes to deploy a microservices application today. You must enumerate the servers, their IP addresses, and the addresses of the microservices they need to talk to, usually in a configuration file [7].

With the GVM the application simply creates a subset of a tree and deploys the application on that subset tree. It doesn't need to know the exact set of cells or even how many are in the set. Task migration is far simpler since addresses don't change if the tasks stay on the tree when they move.

This mechanism enables a simpler way to implement elasticity. A central manager is needed today when expanding or contracting the number of instances of a microservice, largely because of the address management problem. Those decisions can be made locally with tree-based addressing.

Say that 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. Say that the cell to the left of the root in the picture gets overloaded. It can just 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 ms instead of the seconds it takes today.

7 Security

Network vulnerabilities abound. Successful breaches occur on a regular bases. As Marcus Ranum has reportedly asked, "If what we are doing was effective, shouldn't we be seeing some improvement?"

Ariadne makes entire classes of attacks inexpressible. For example, a successful breach of an application running in a datacenter is often leveraged into an escalation of privilege attack against the operating system. Once that's done, the attacker can send malformed packets on the network causing the switches to go into learning mode. Watching that 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 [9].

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 turns off any ports connected to other tenants. As far as the tenant is concerned, the corresponding links don't 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 CPU on the NIC. Hardware and side channel attacks mounted from applications or the host operating system don't reach that CPU.

8 Other Work

People recognize that the network is causing problems for distributed applications, and there are many attempts to address them. The most significant is

Software Defined Networks [10], which does in software what has traditionally been done in custom hardware. The added flexibility makes it possible to address problems in a way that meets an individual customer's needs. In addition, new features can be applied far faster than waiting for the next hardware version.

Lately, there have been a number of smart NICs that offload dealing with network issues from the main CPU, such as Nitro [11] and Snap [12]. Others, notably Pensando [13] and Fungible [14], move parts of the application to the smart NIC.

None of these approaches make any fundamental changes to the network itself or its protocols. While they do a wonderful job treating the symptoms, none of them addresses the root causes.

InfiniBand (IB) [15] is a network and set of protocols that provides reliable, in-order delivery of packets at low latency. It's switches use credit-based flow control on multiple *virtual lanes* to support quality of service guarantees. The IB protocol is not ping pong, so it can't implement a common knowledge protocol.

9 Current Status

We have spent a lot of time exploring the design space. Different tree building algorithms produce trees with different properties, which we now know how to control. There are many options for picking the next port to try when failing around a broken link. We now understand which ones give good best-case behavior and which we should use when optimizing for best worst-case behavior. The more metadata we collect, the better failure recovery works, but the longer basic tree building takes. We have quantified the trade-offs.

All the algorithms discussed so far have been implemented separately. We have two servers that implement the Liveness and Common Knowledge protocols in a Linux driver. A simulator written in JavaScript was used to verify the tree building and failover mechanisms shown in Figure 3. A Rust emulator implements the basic messaging and packet forwarding mechanisms as well as tree subsetting and a model of how to deploy applications and have them talk to each other.

Each of these implementations has its own technical debt. At the time of this writing, we are actively seeking funding to hire engineers and buy enough servers to provide an Alpha version that potential customers can use to validate the benefits we claim.

10 Conclusions

The datacenter networks we use today have served us well for decades, but the design is showing its age. Ariadne is fundamentally different, starting with a different topology, changing to a new set of link protocols, and providing an addressing scheme better suited to today's applications. It even takes an approach to orchestration that better fits the model of distributed applications.

A key design goal of Ariadne was to fix in the network problems that occur in the network. Ariadne does that so quickly that applications see a network that never appears to break. Legacy mode tunnels TCP/IP over the Ariadne network, so unmodified applications can get some benefit. Also, you don't need to build a new datacenter. An Ariadne rack or two in the middle of an existing datacenter can run some key applications.

The real magic happens for those developers who code to Ariadne's network model. We believe that many uses of heavyweight protocols, such as consensus and two-phase commit, can be replaced by common knowledge across a link. Applications that take advantage of that feature will be qualitatively better for it.

11 ACKNOWLEDGMENTS

We'd particularly like to thank the many people who suffered through earlier explanations of what became Ariadne. Invaluable feedback was provided by Carlos Baquero, Paulo Almeida, Jamie Vicary, and Mark Burgess, as well as members of our team, Bill Jackson and Bruce Borden. Douglas Crockford's careful reading was a significant help.

References

[1] Netronome, https://www.netronome.com/products/nfe/

[2] J. Y. Halpern and Y. Moses. 1990. "Knowledge and common knowledge in a distributed environment." J. ACM 37, 3 (July 1990), 549?587. DOI:https://doi.org/10.1145/79147.79161

[3] J. Gray. "Notes on database operating systems," IBM Res. Rep. RJ 2188. IBM, Aug. 1987. https://cs.nyu.edu/courses/fall18/CSCI-GA.3033-002/papers/Gray1978.pdf, pg. 73.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson. 1985. "Impossibility of distributed consensus with one faulty process." J. ACM 32, 2 (April 1985), 374?382. DOI:https://doi.org/10.1145/3149.214121

[5] D. Skeen, "A Quorum-Based Commit Protocol," Technical Report, Cornell U., 1982, https://ecommons.cornell.edu/handle/1813/6323

[6] T. Treat, "You Cannot Have Exactly-Once Delivery", https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/, 2015

[7] Kubernetes, https://kubernetes.io/docs/concepts/

[8] Leslie Lamport. 1998. "The part-time parliament." ACM Trans. Comput. Syst. 16, 2 (May 1998), 133?169. DOI:https://doi.org/10.1145/279227.279229

[9] I. Ghafir and V. Prenosil, "Advanced Persistent Threat Attack Detection: An Overview," International Journal of Advancements in Computer Networks and Its Security, Volume 4 : Issue 4, 2014

[10] Open Networking Foundation, https://www.opennetworking.org/sdn-definition/

[11] Nitro, https://aws.amazon.com/ec2/nitro/

[12] M. Marty, *et al.* 2019. "Snap: a microkernel approach to host networking." In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP ?19). Association for Computing Machinery, New York, NY, USA, 399?413. DOI:https://doi.org/10.1145/3341301.3359657

[13] Pensando https://pensando.io/our-platform/

[14] Fungible, https://go.fungible.com/clients/fungible/uploads/Unlocking-True-Performance-and-Cost-Efficiencies-of-Storage.pdf

[15] InfiniBand, Volume 1 Architecture Specification, Release 1.4, https://cw.infinibandta.org/document/dl/8567, April 2020 and Volume 2 Architecture Specification, Release 1.4, https://cw.infinibandta.org/document/dl/8566, April 2020