Prototypes, Papers, and Production

Developing a globally distributed purging system

Bruce Spang

@brucespang

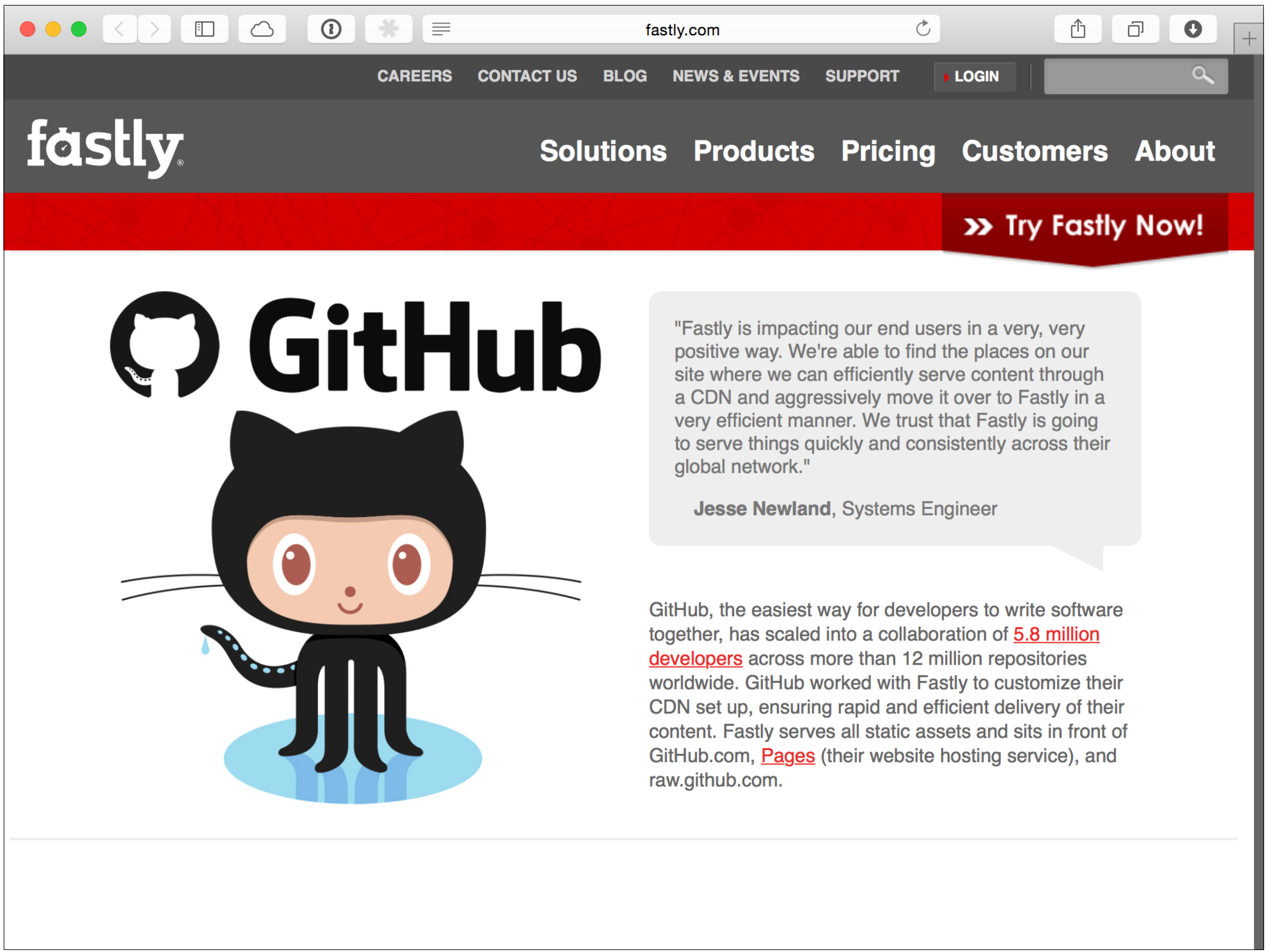
Tyler McMullen

@tbmcmullen

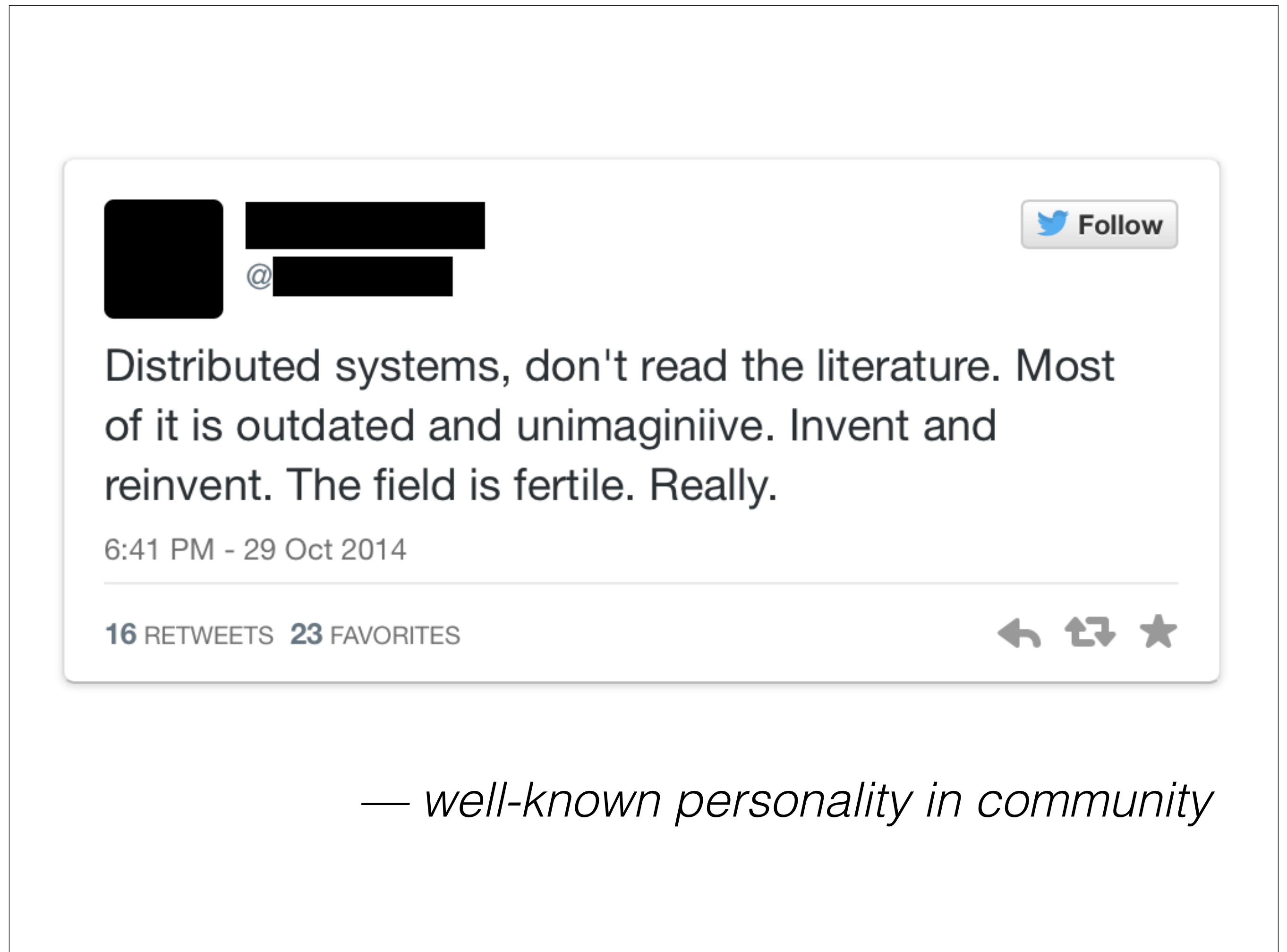
fastly®

What is a CDN?

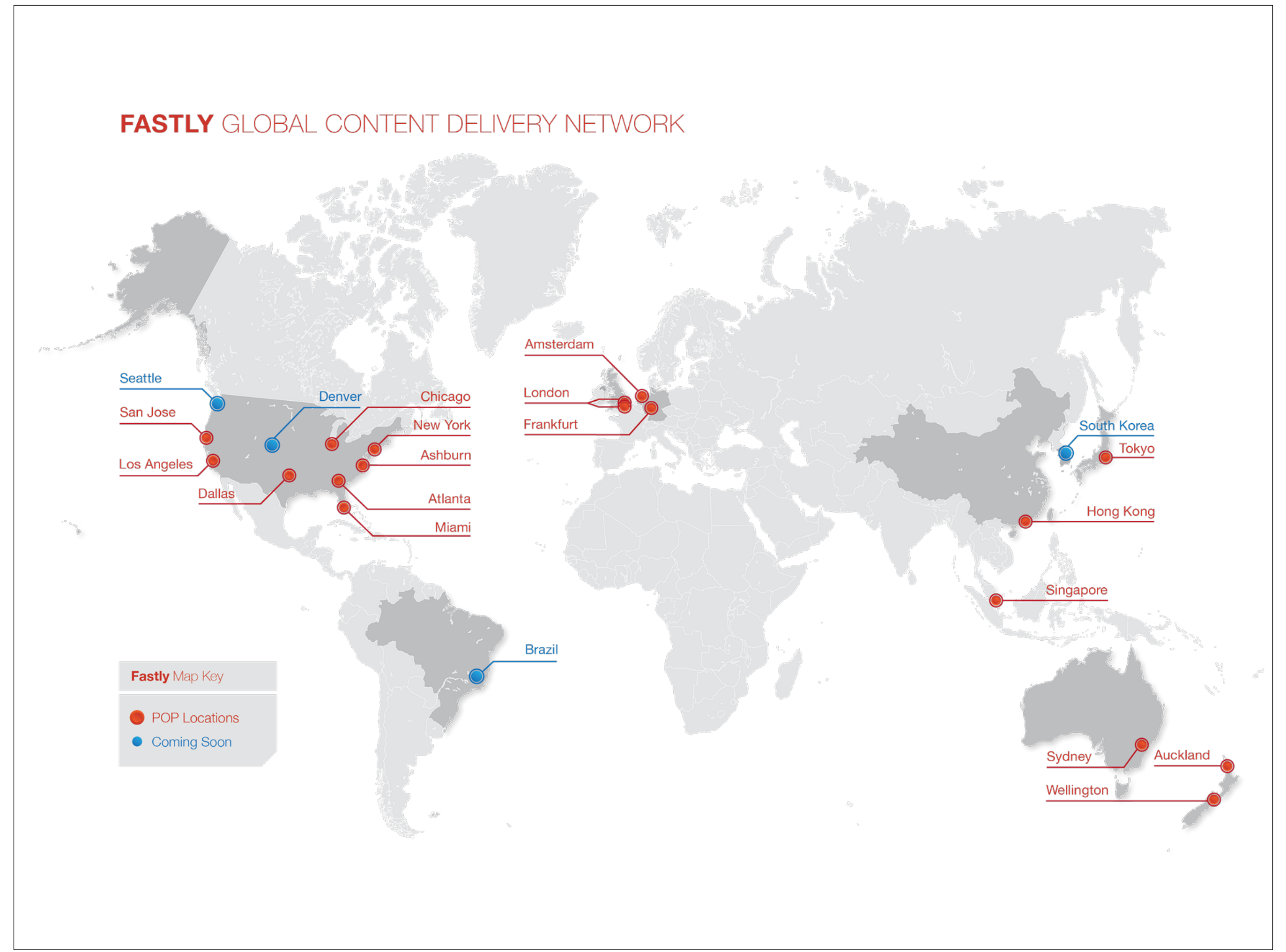
You probably already know what a CDN is, but bear with me. A CDN is a “Content Delivery Network”. It’s a globally-distributed network of servers and at it’s core the point is to make the internet better for everyone who doesn’t live across the street from your datacenter. You might use it for images, APIs, …



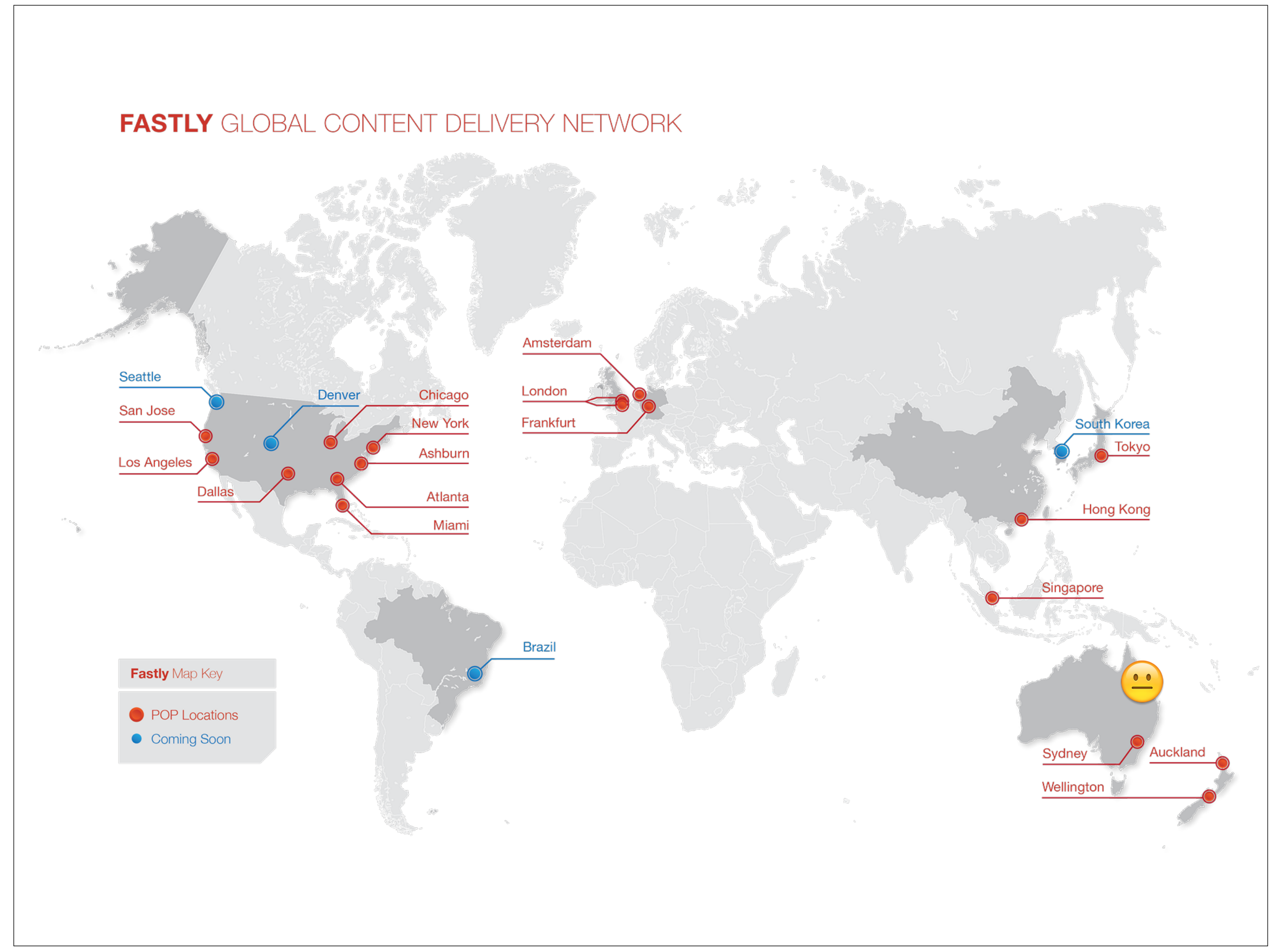
Or websites. For instance, this website about how much GitHub loves Fastly… (Don't worry, this is the last slide that is anything at all resembling a sales pitch.)



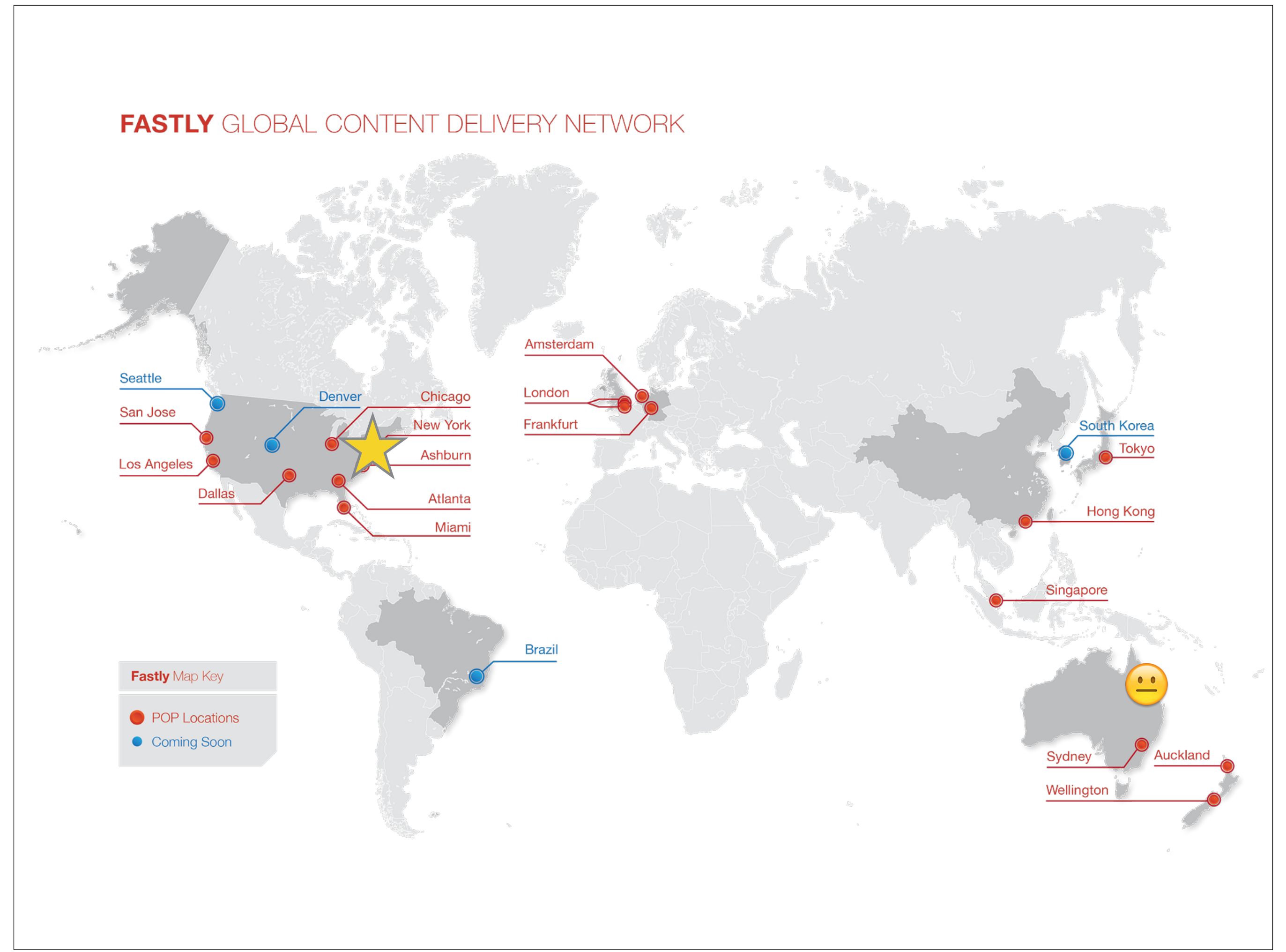
Or even this tweet of terrible advice. This tweet becomes more relevant as we go along…



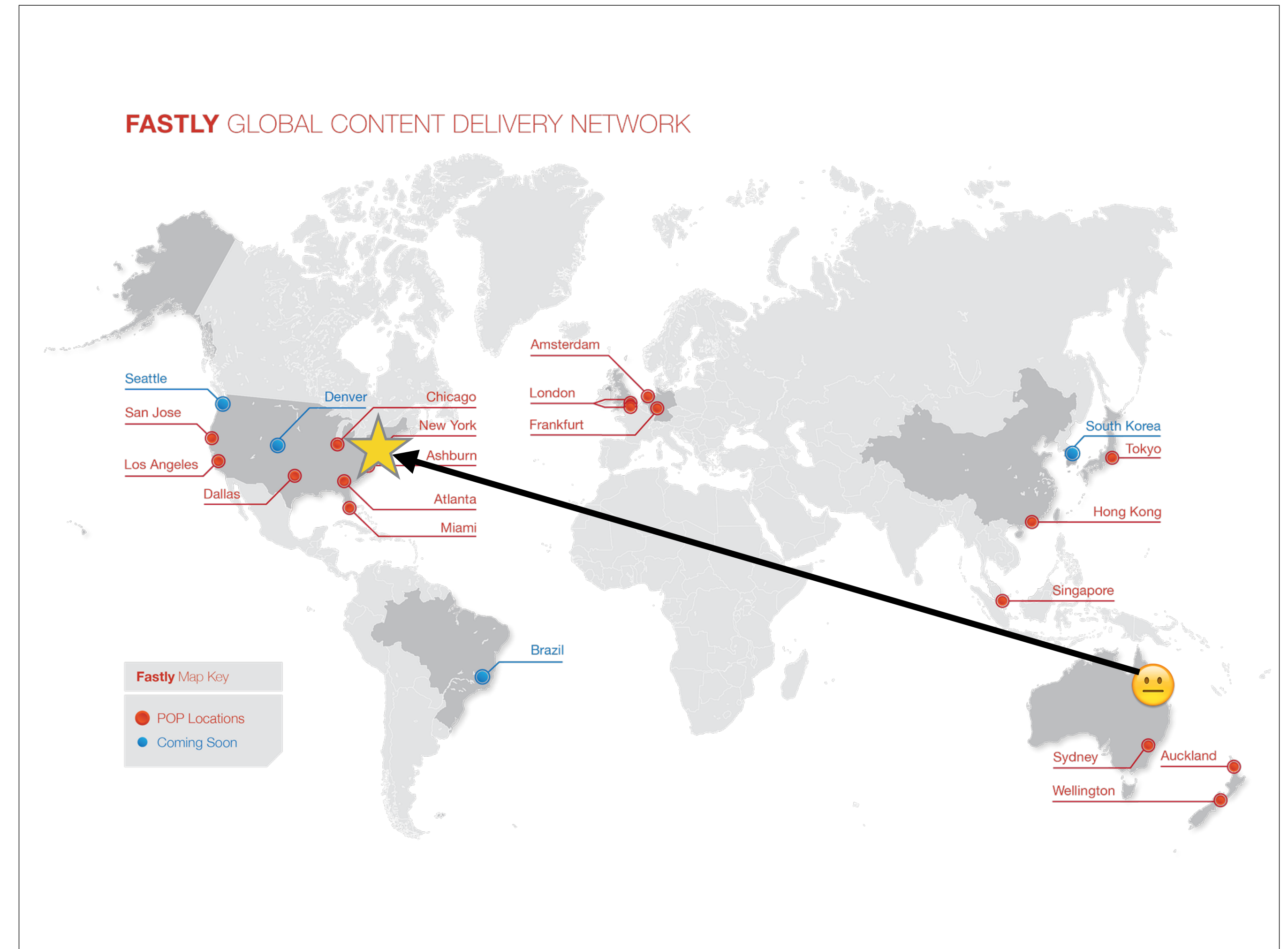
So, our goal is to deliver whatever your users are requesting as quickly as possible. To do this, we have a network of servers all over the world which cache content.



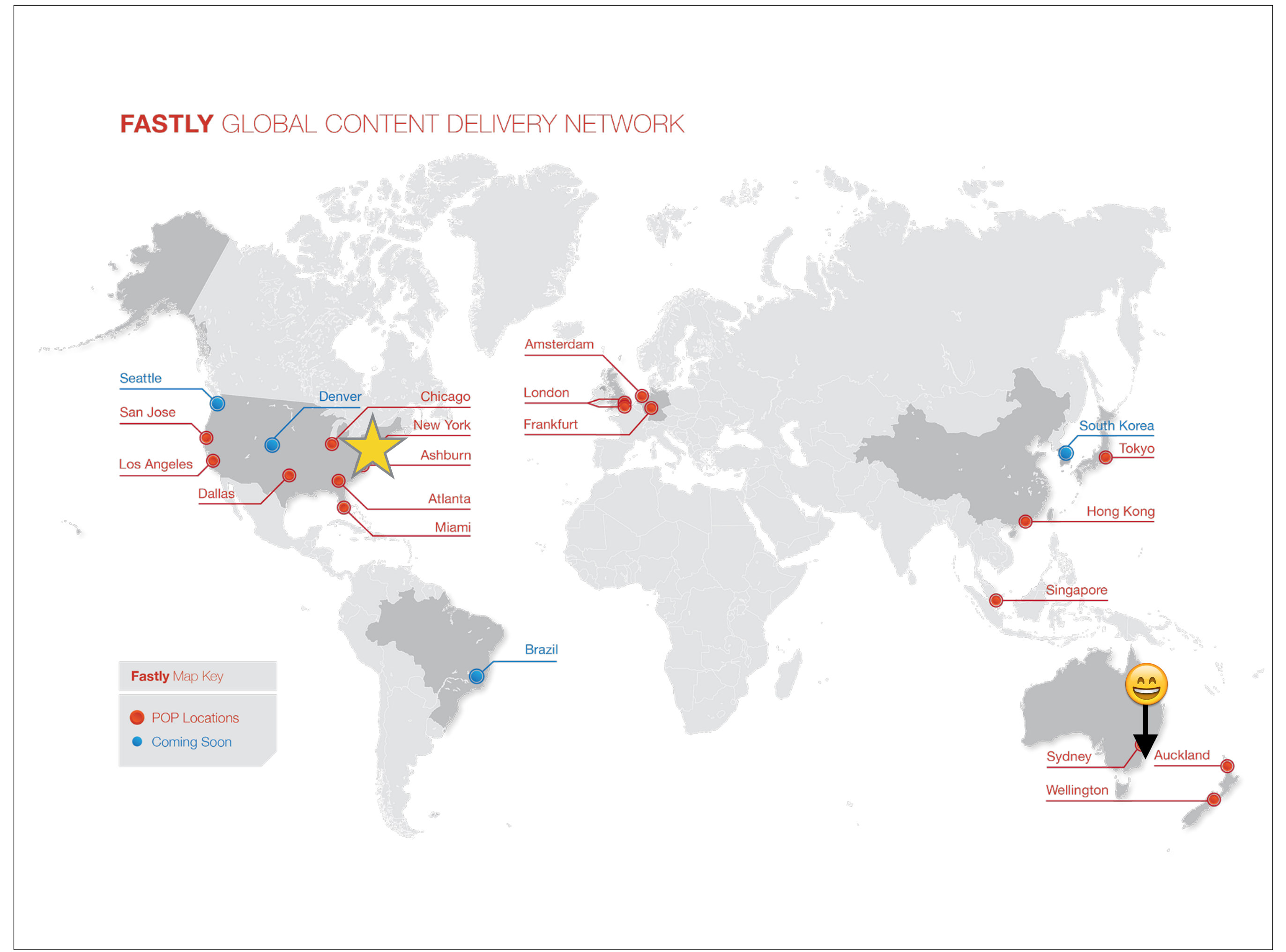
Suppose you live in Australia



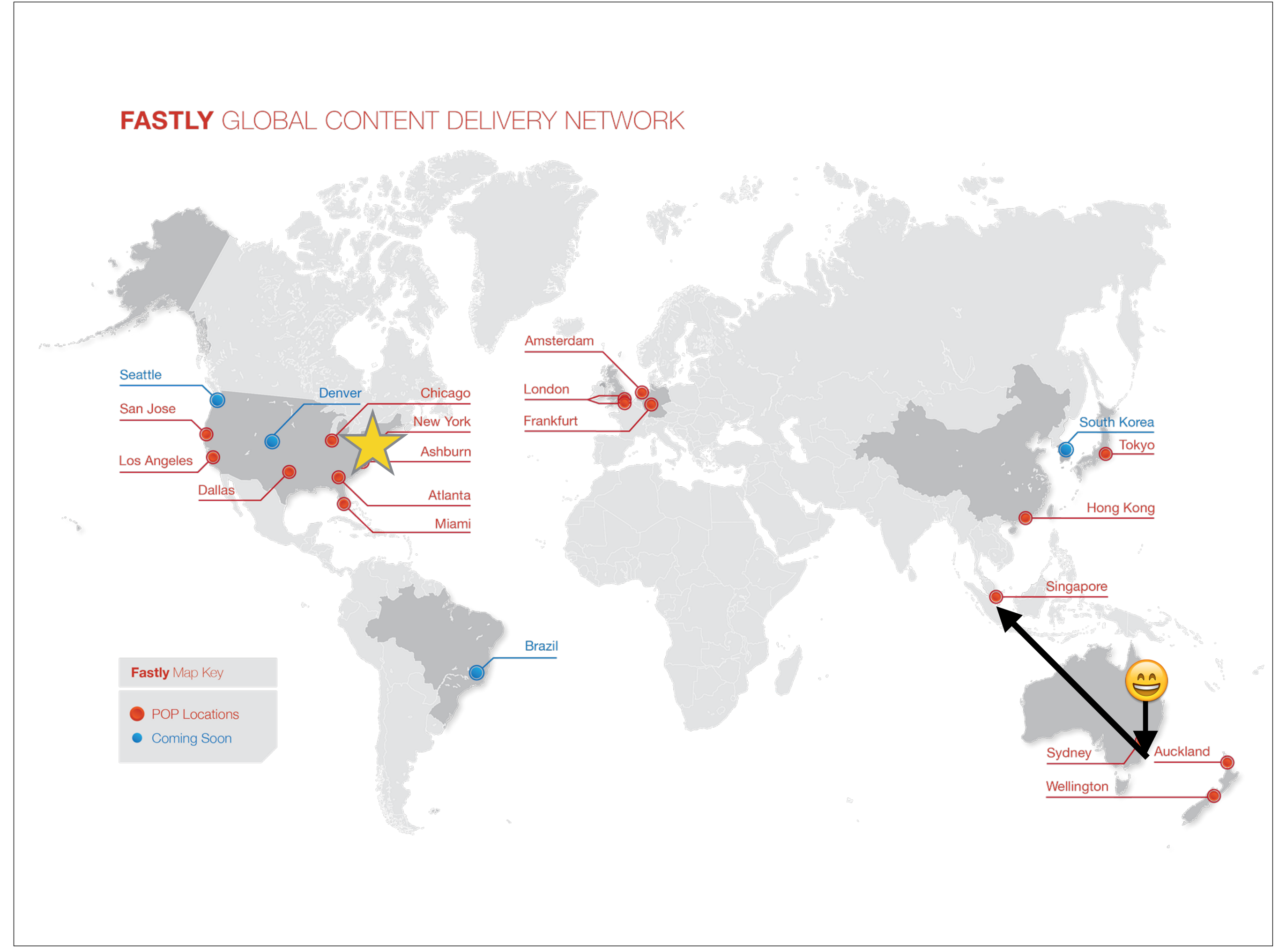
and you want to visit a site which is hosted on servers in New York



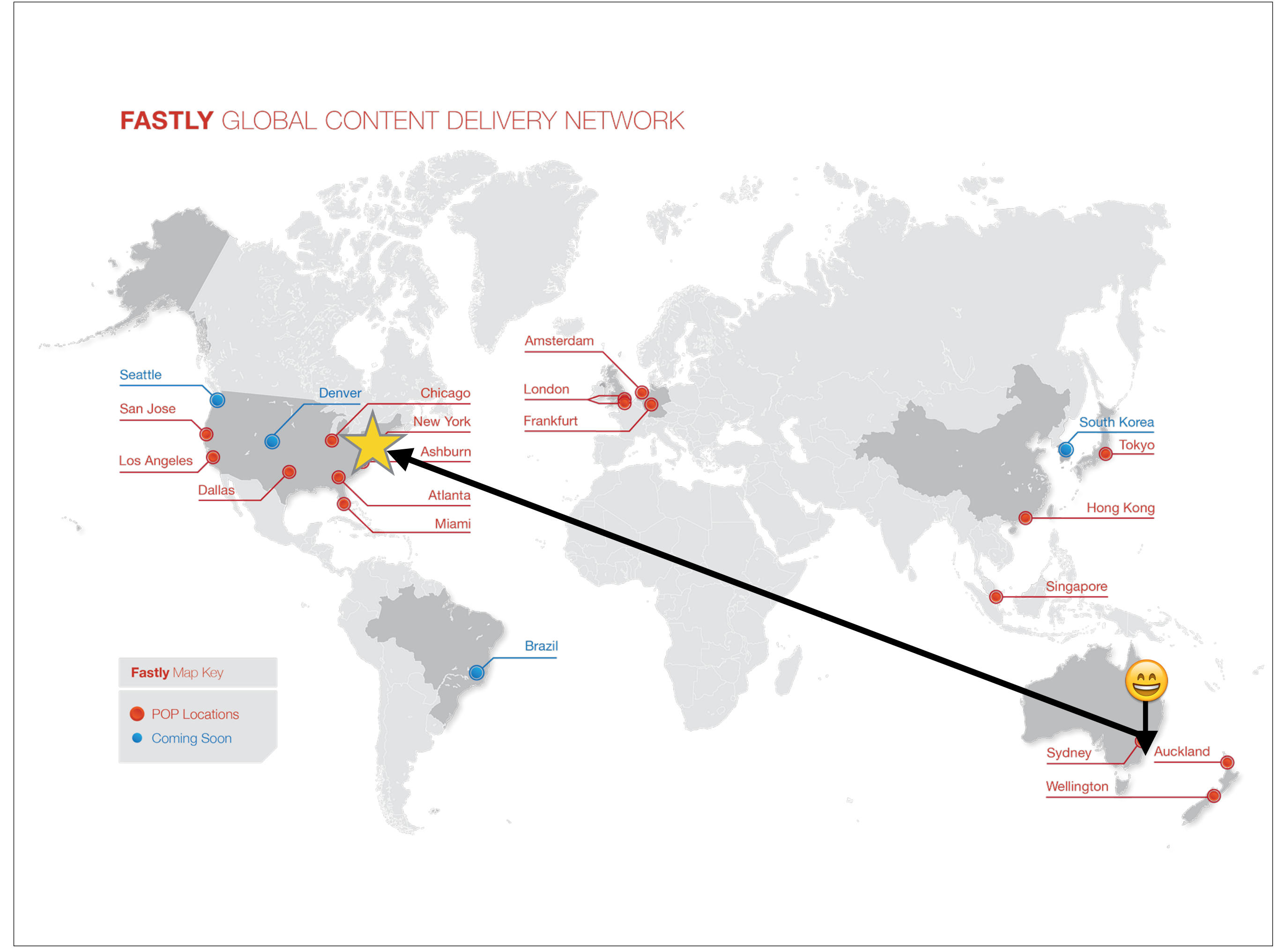
normally, you would go directly to this site half way around the world, and it would take some time. Note that this is greatly simplified, as your request would likely bounce between 20 or 30 routers and intermediaries before getting to the actual server.



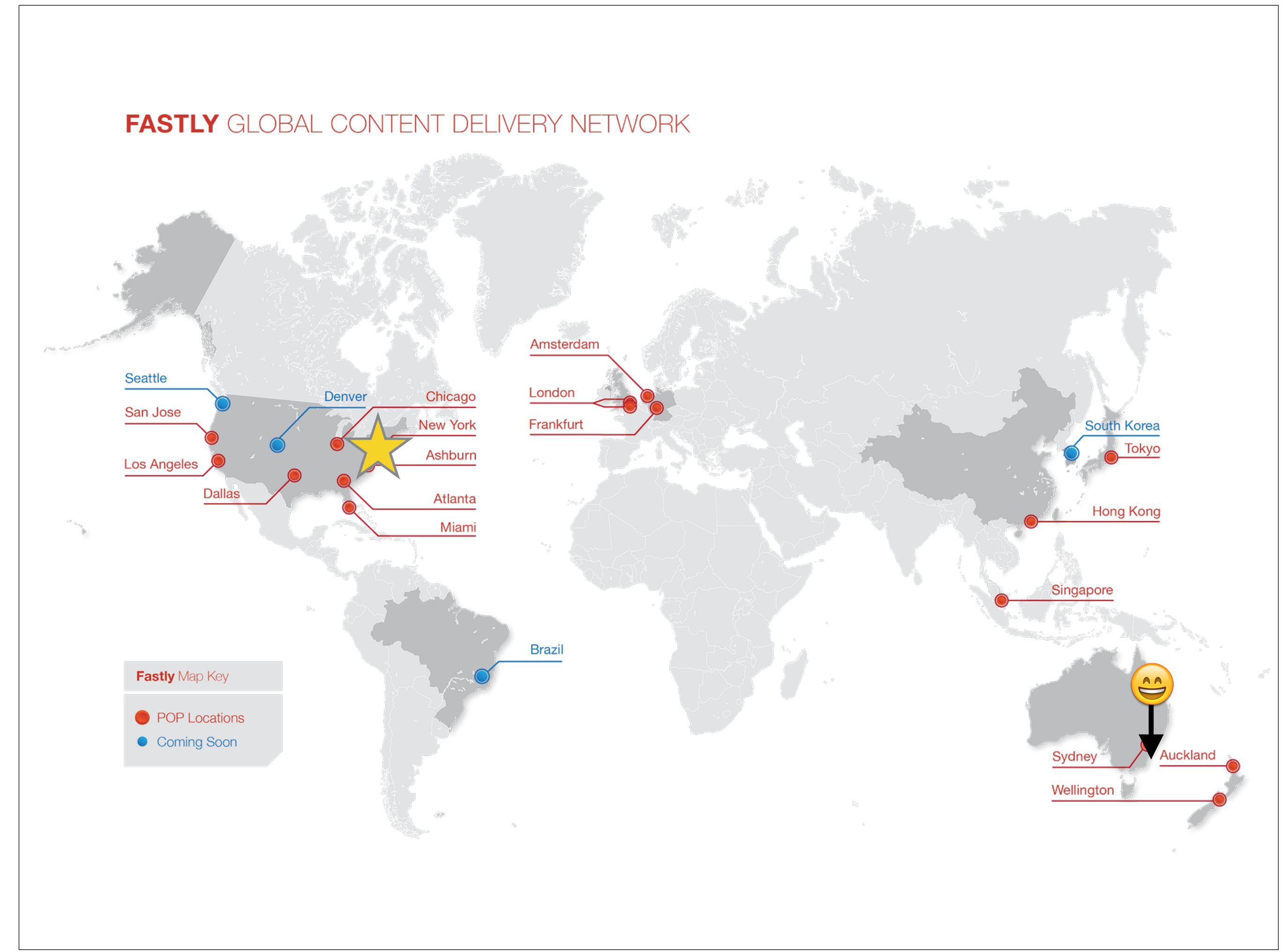
with fastly, instead you would go to one of our servers in say, Sydney. normally, a copy of the website would be on that server, and it would be much faster.



If the content isn’t already there, we could request it from other local servers.



But ultimately, if it’s a new piece of content, you may still have to make a request to New York.



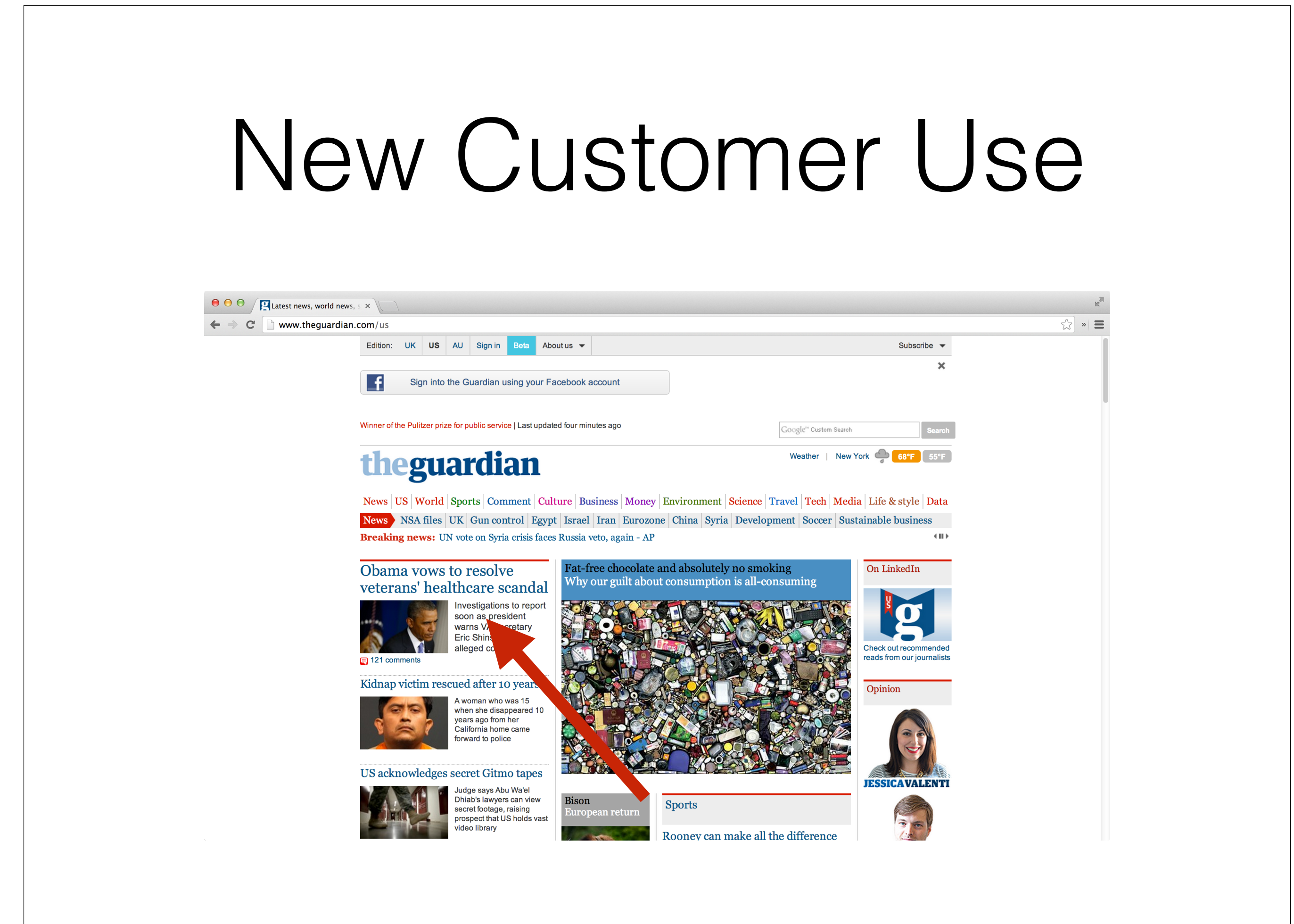
However, next time you or someone else visits the site, it would be stored on the server in sydney, and would be much faster.

Cache Invalidation

however, once a site is stored on a server, you might want to remove it for some reason; we call this a purge.

for example, you might get a DMCA notice and have to legally take it down.

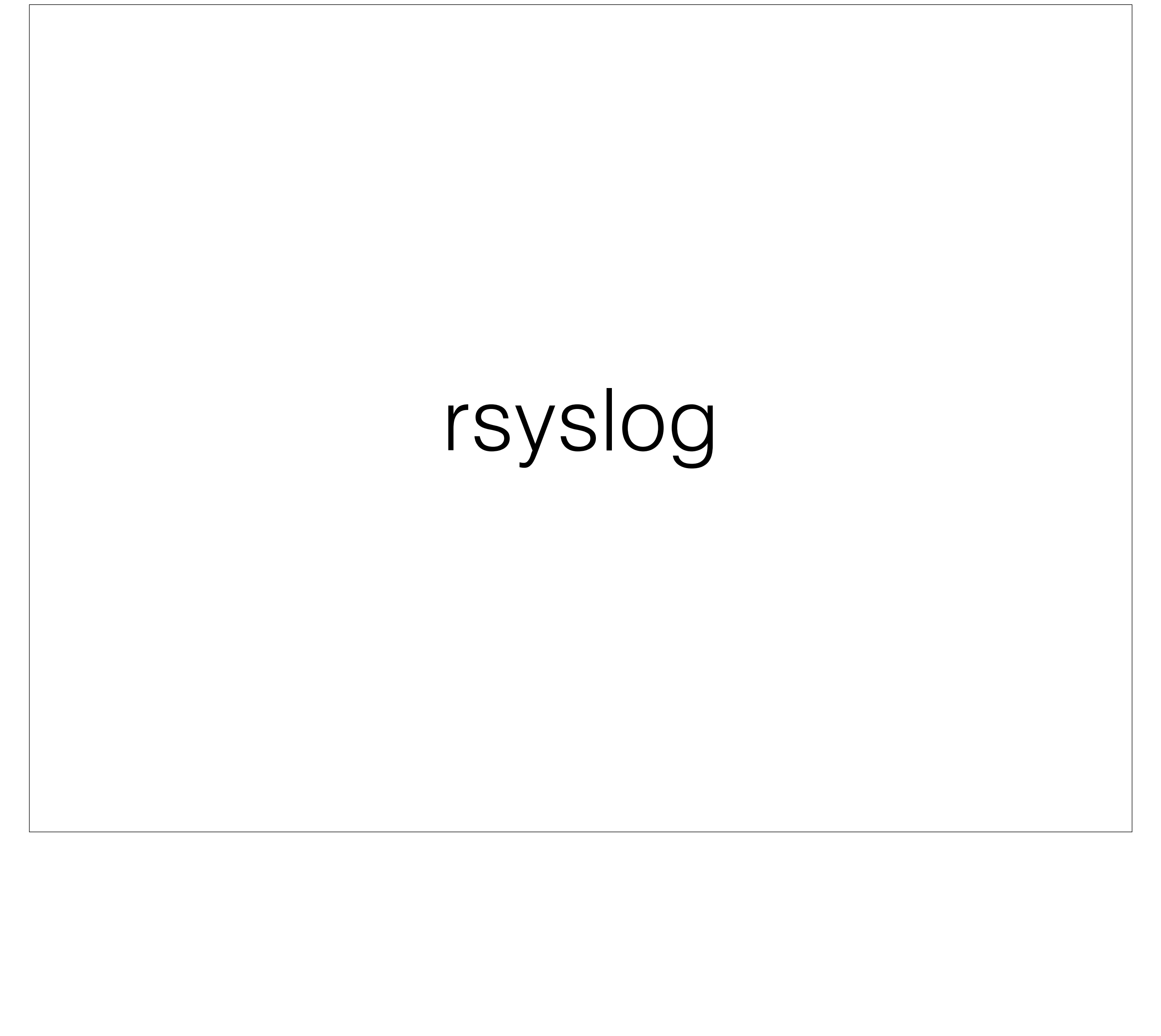
Or even as something as simple as your CSS or an image changing.

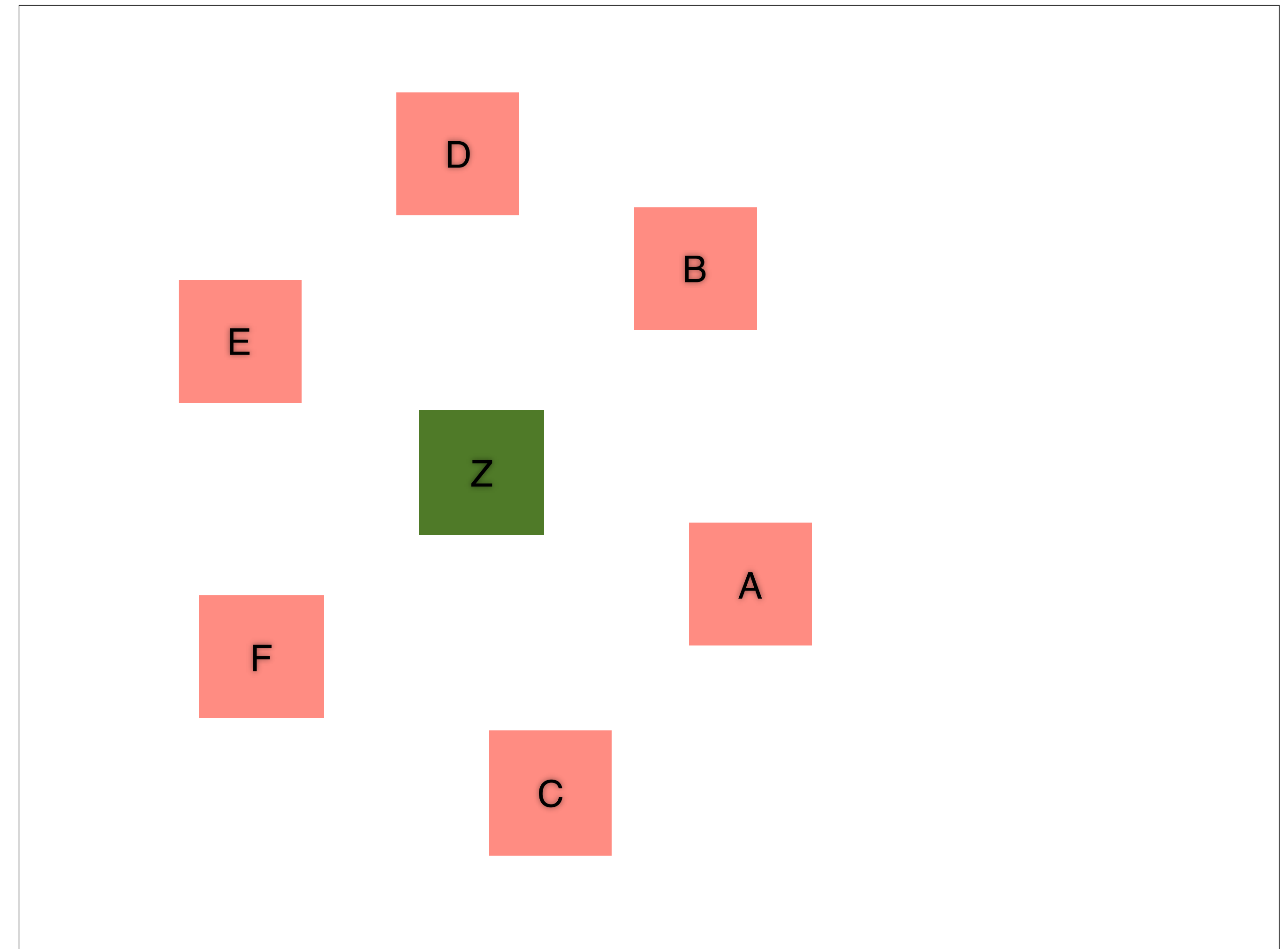


One of the points of Fastly though, from the very beginning, was making it possible to purge content quickly. For instance, The Guardian is caching their entire homepage on Fastly. When a news story breaks, they post a new article, and need to update their homepage as quickly as possible.
That purge needs to get around the world to all of our servers quickly and reliably.

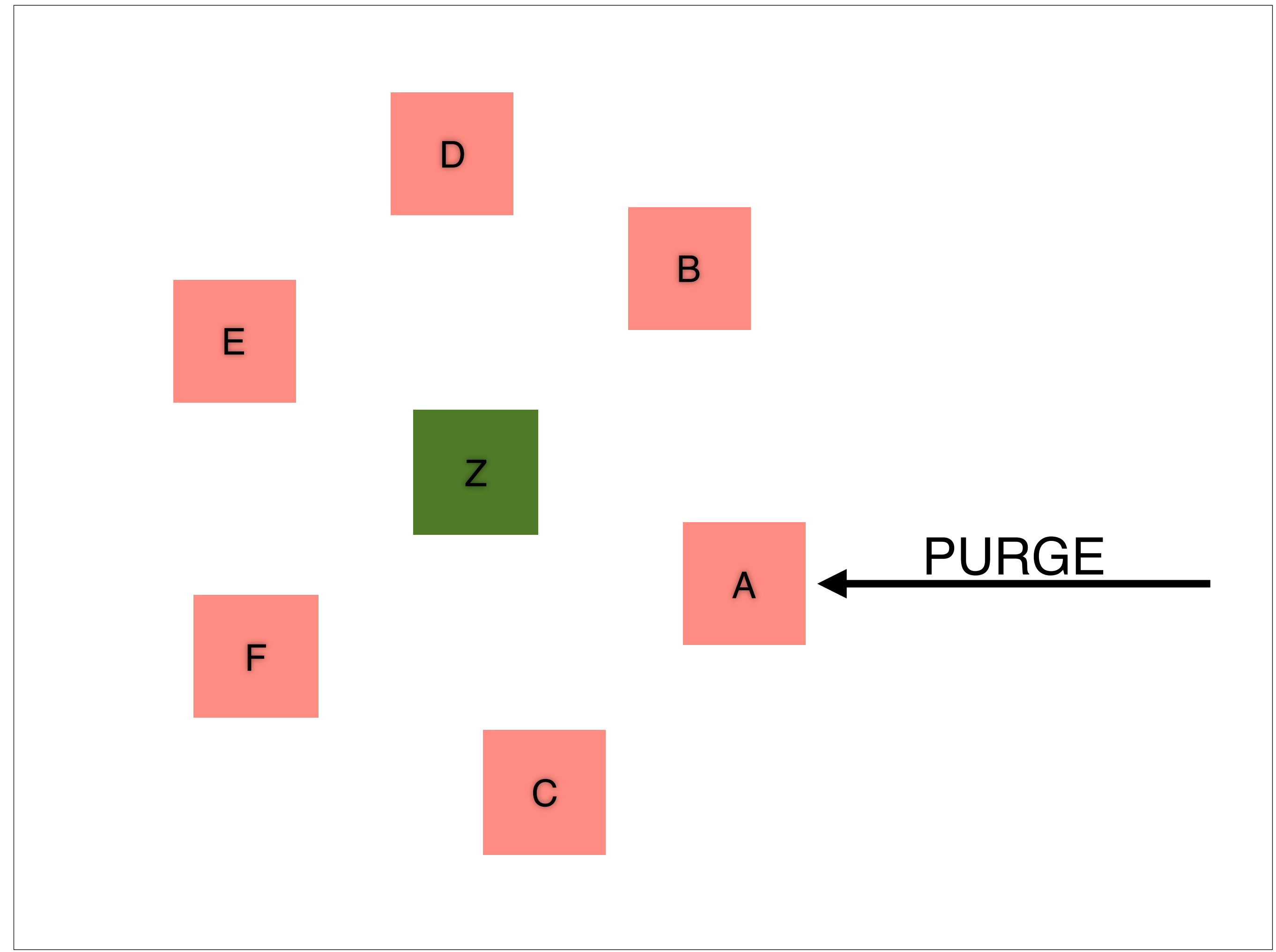
Step One

Make it

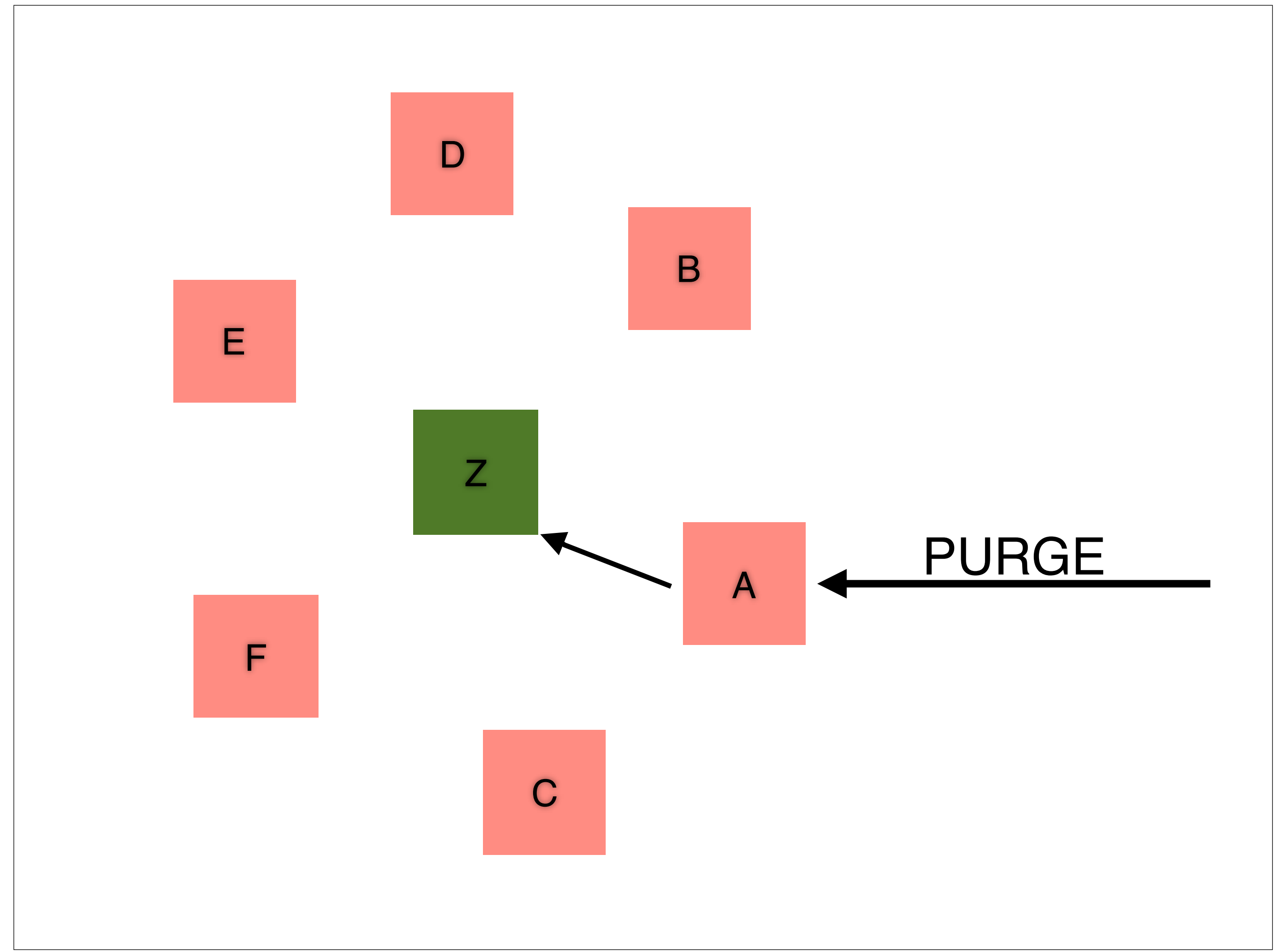

rsyslog



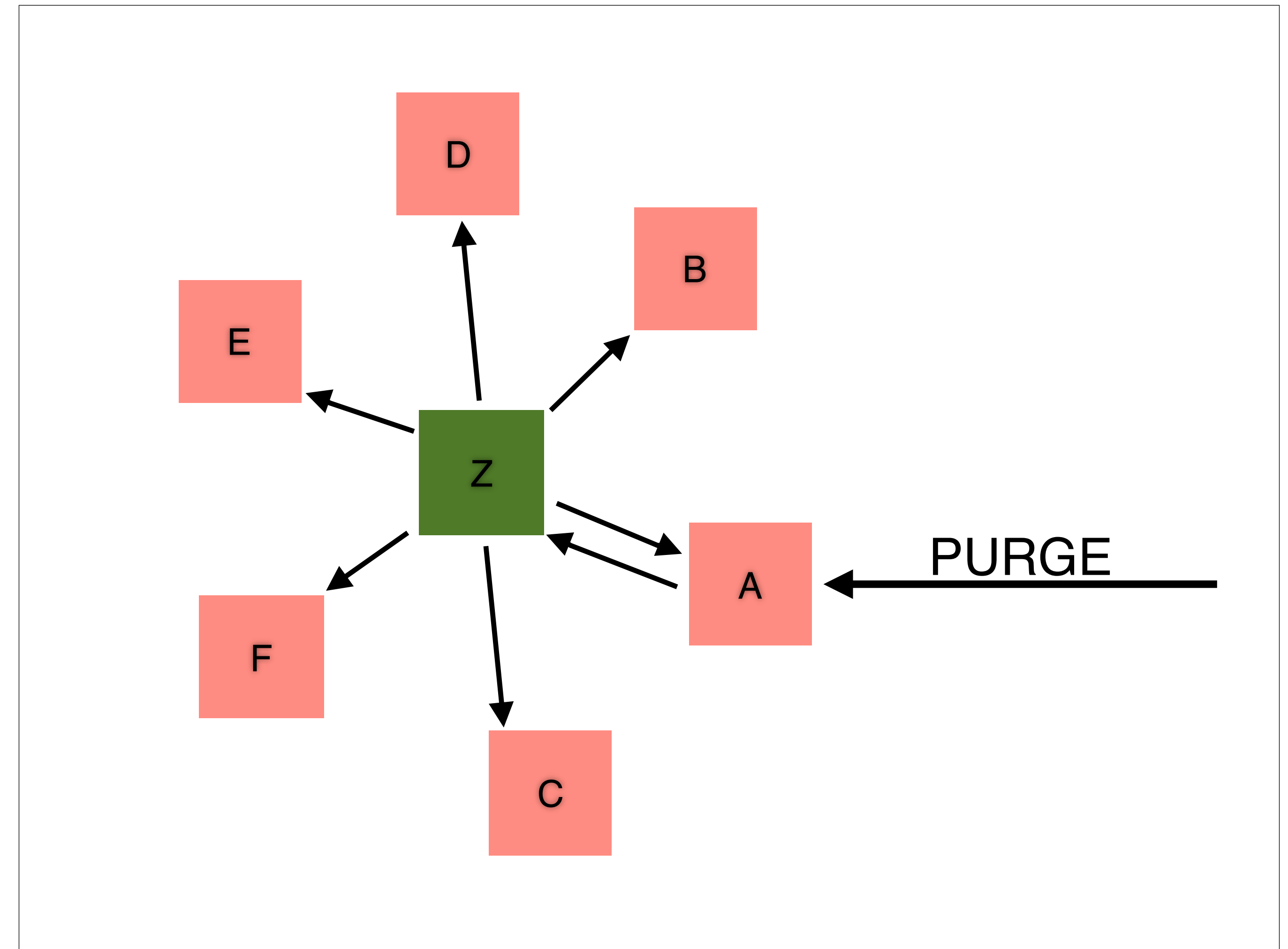
So, here’s how it works. We have a bunch of edge nodes spread around world. A might be in New Zealand. F could be in Paris.



A purge request comes in to A. The purge could be for any individual piece of content.



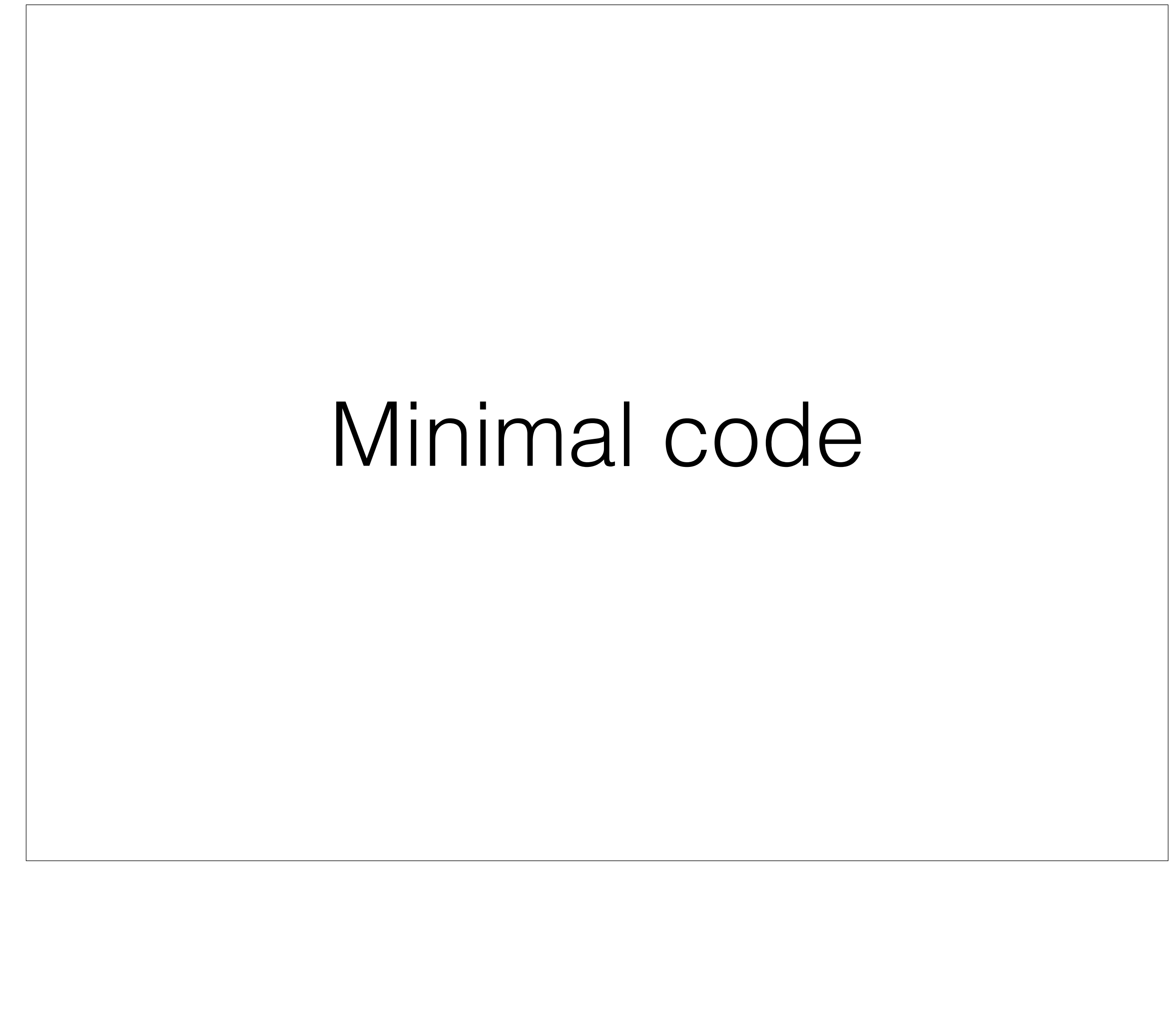
A forwards it back to our central rsyslog “broker” of sorts, Z. Which might in, say, Washington DC.



And the broker sends it to each edge node.

It also probably looks pretty familiar. It’s really the simplest possible way of solving this problem. And for a little while it worked for us.

Already deployed

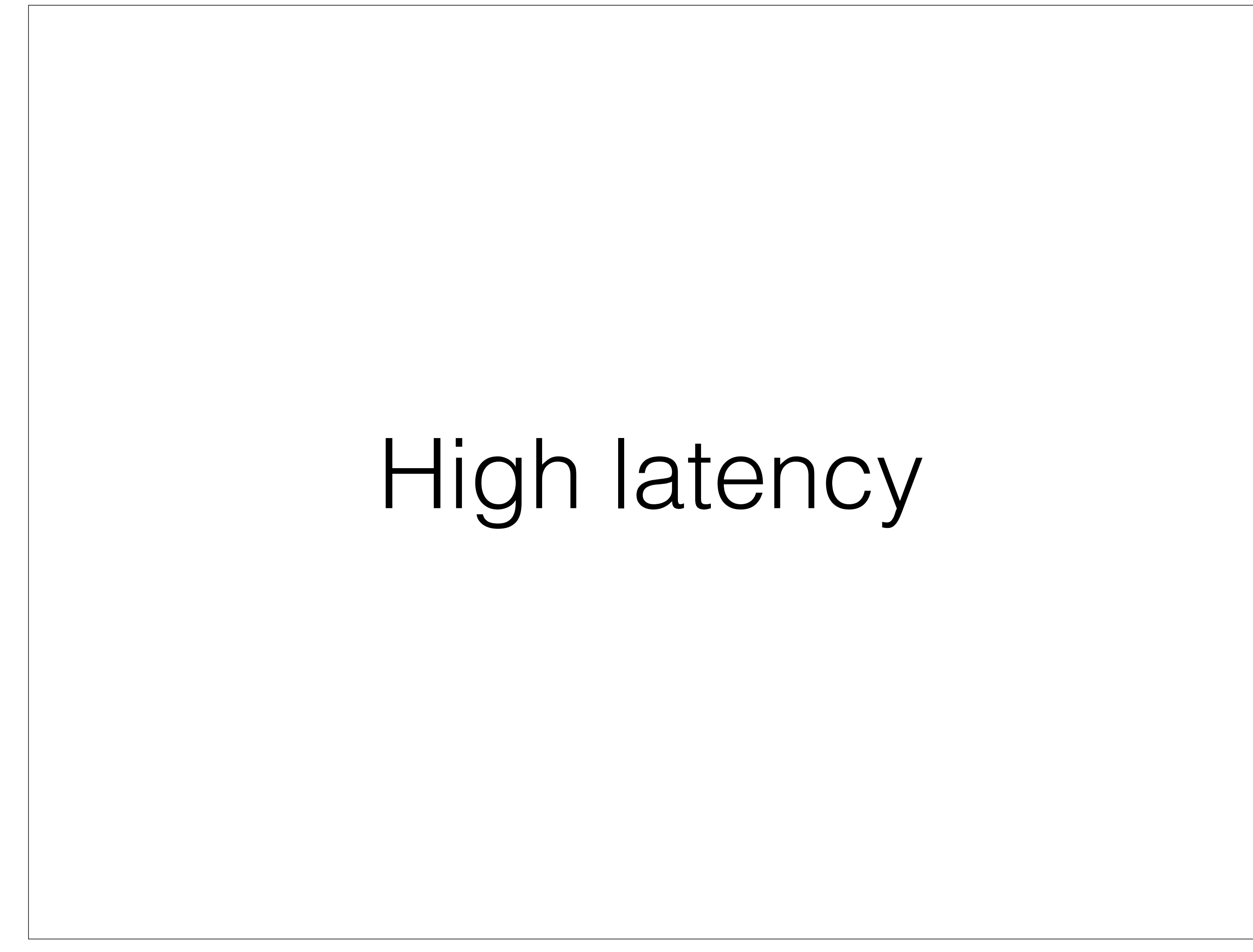

Minimal code

The way Rsyslog works is trivial to reason about.

That also means that it's really easy to see why this system is ill-suited for the problem we're trying to solve.

At its core, it's a way to send messages via TCP to another node in a relatively reliable fashion.

Why does it fail?



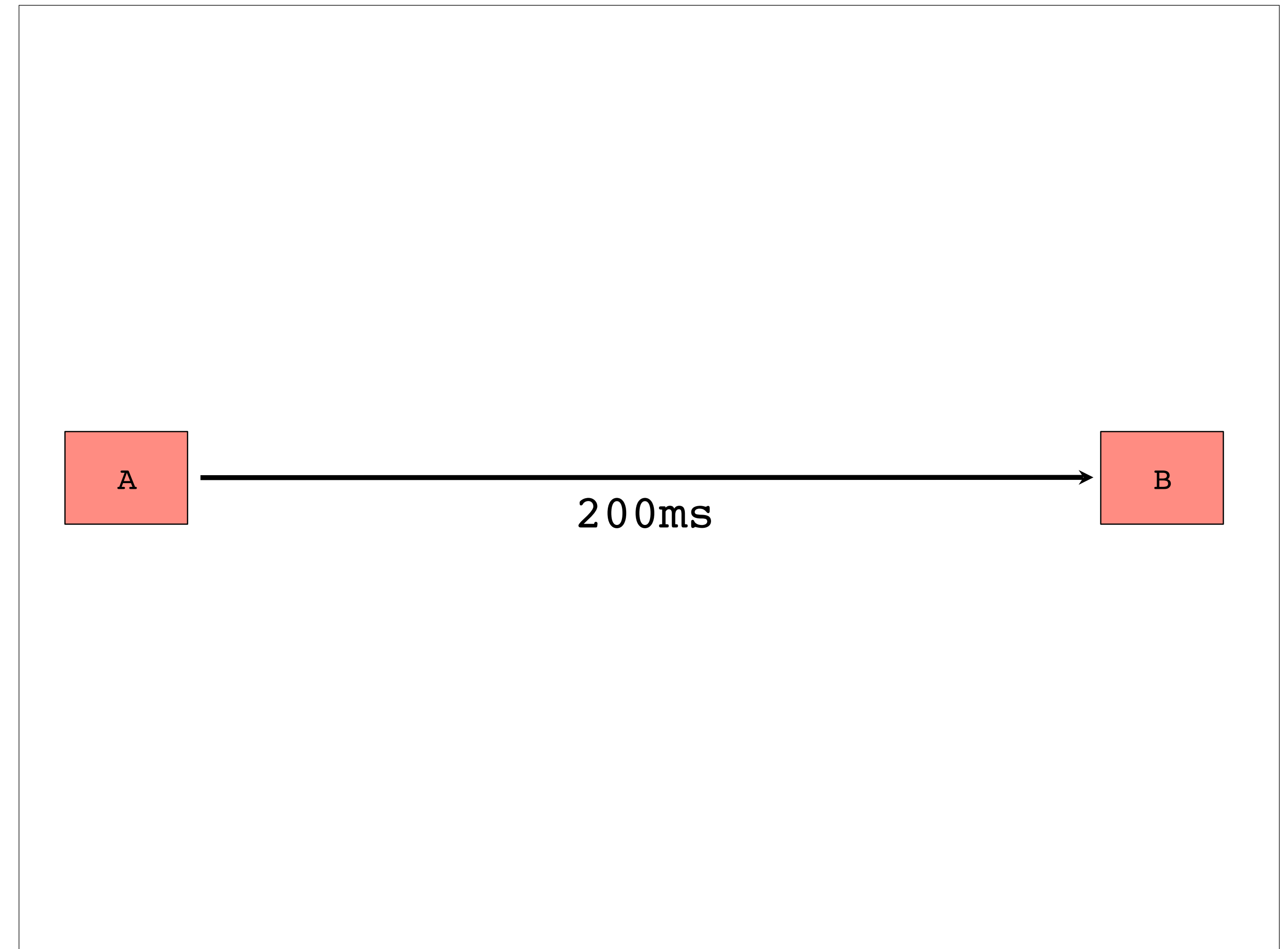
Two servers sitting right next to each other, would still need to bounce the message through a central node in order to communicate with each other.

Obvious and enormous SPOF in the central node

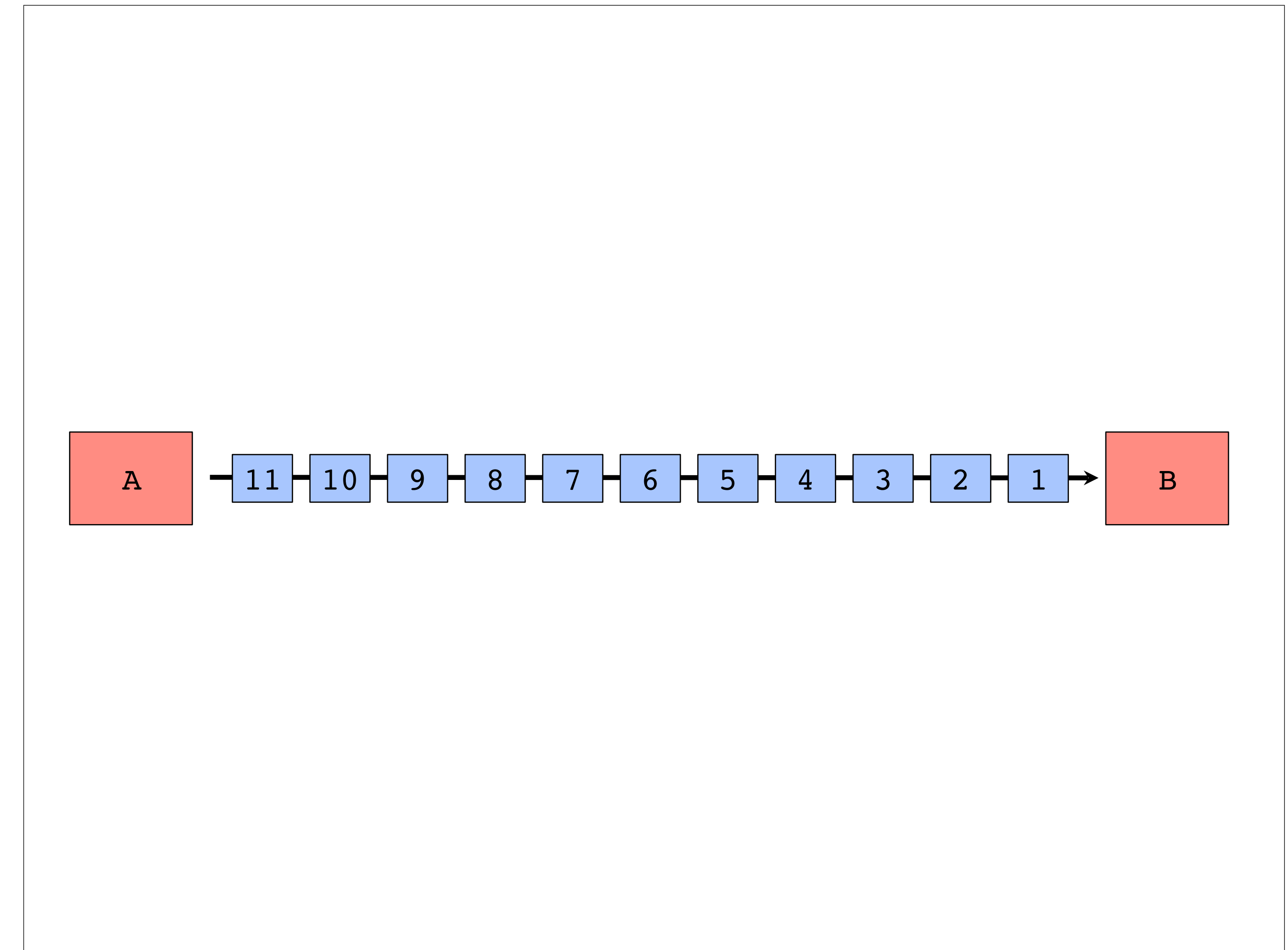
This system has stronger consistency guarantees than we actually need.

For instance, this system uses TCP and thus guarantees us in-order delivery.

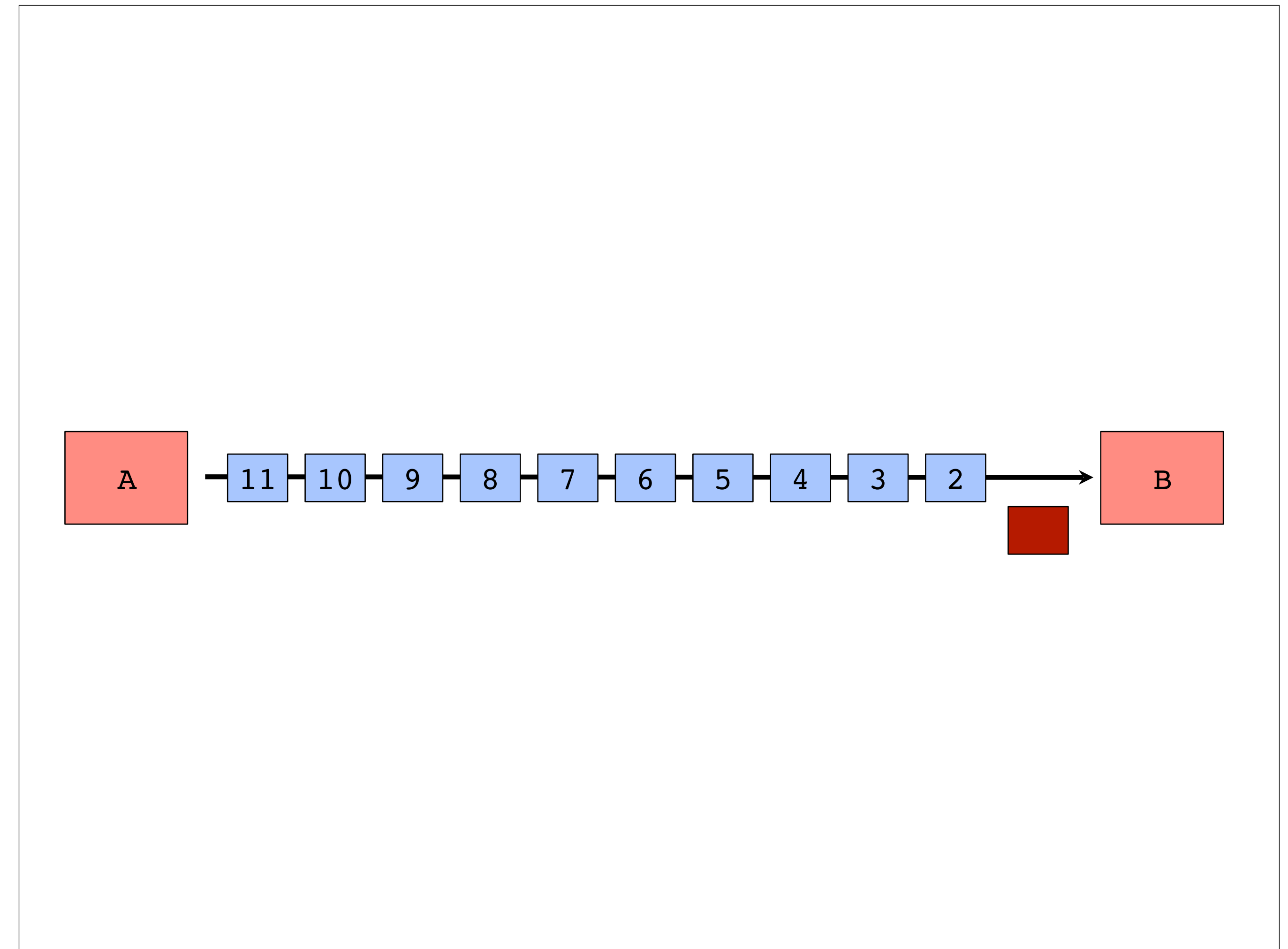
How does that actually affect the behavior in production?



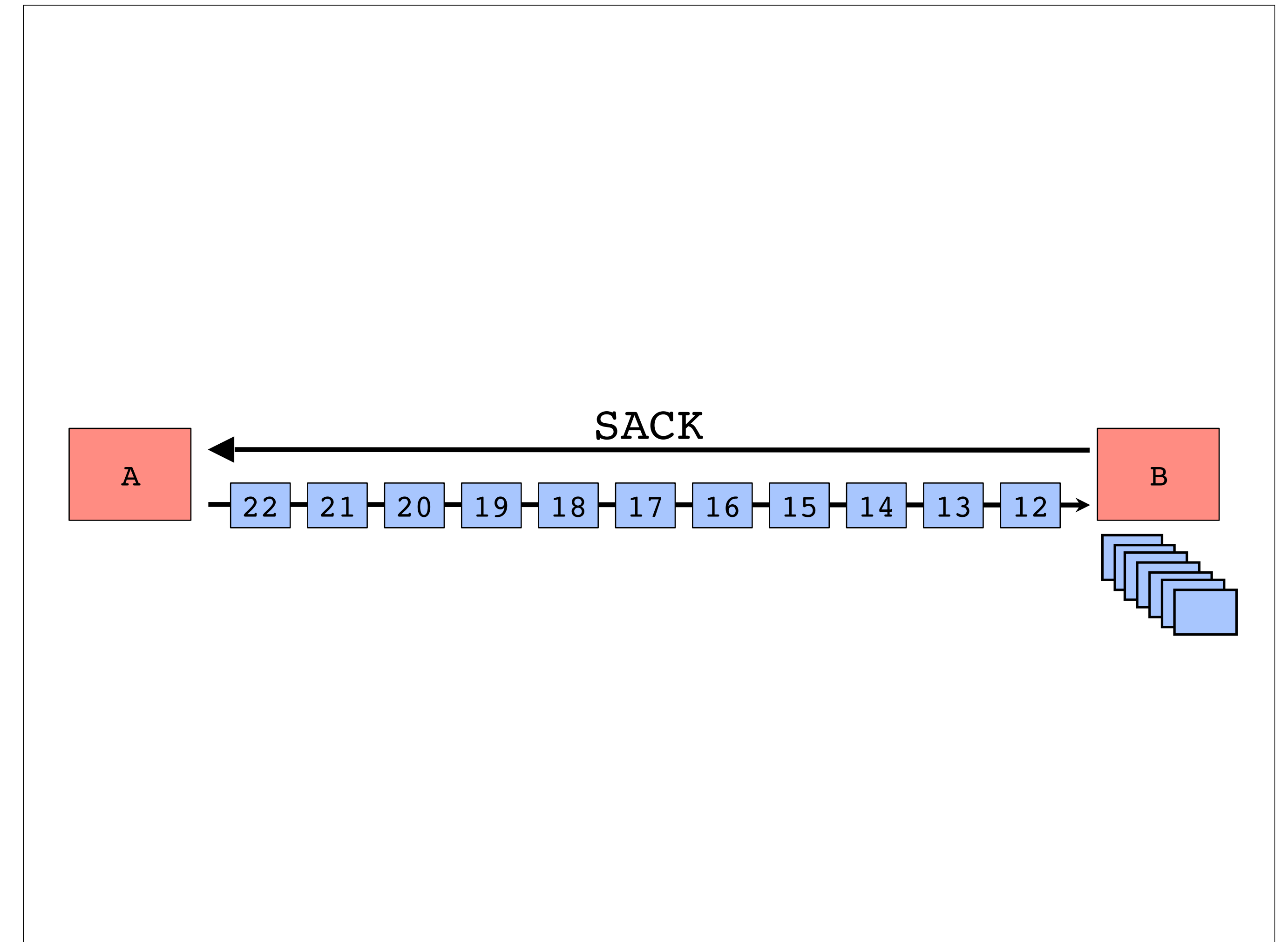
Let’s say we’re sending 1000 messages per second. One message every millisecond. Let’s say the node we’re sending to is 200ms away



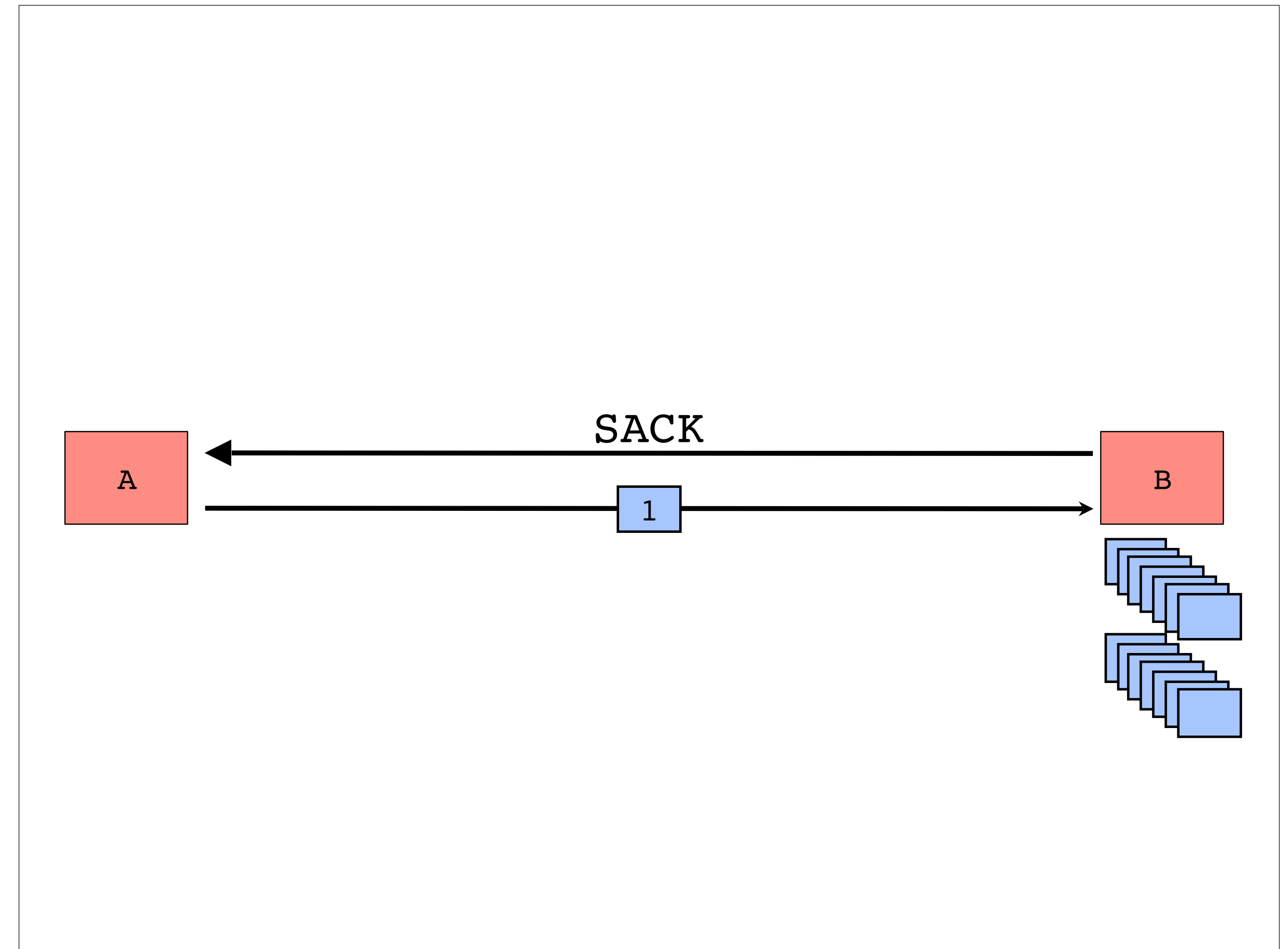
That means that at any time there are ~200 messages on the wire.



Let’s say a packet gets dropped at the last hop. Instead of having one message be delayed, what actually happens is the rest of the packets get through but are buffered in the kernel at the destination server and don’t actually make it to your application yet.



The destination server then sends a SACK (which means “Selective Acknowledgement”) packet back to the the origin. Which effectively says, “Hey I got everything from packet #2 to packet #400, but I’m missing #1.”. While that is happening, the origin is still sending new packets which are still being buffered in the kernel.



Then finally, the origin receives the SACK and realizes the packet was lost, and retransmits it.

So, what we end up having is 400ms of latency added to 600 messages.
- 240,000ms of unnecessary delay

Each of those could have been delivered as they were received. We and our customers would have been just as happy with that. But instead they were delayed. Thus, this is the wrong consistency model.

Make it Interesting

Atomic Broadcast

read papers on Atomic Broadcast, because it seemed like the closest fit to what we're trying to do

Zab: High-performance broadcast for primary-backup systems

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini
Yahoo! Research
{fpj,breed,serafini}@yahoo-inc.com

***Abstract*—Zab is a crash-recovery atomic broadcast algorithm we designed for the ZooKeeper coordination service. ZooKeeper implements a primary-backup scheme in which a primary process executes clients operations and uses Zab to propagate the corresponding incremental state changes to backup processes[1]. Due the dependence of an incremental state change on the sequence of changes previously generated, Zab must guarantee that if it delivers a given state change, then all other changes it depends upon must be delivered first. Since primaries may crash, Zab must satisfy this requirement despite crashes of primaries.**

Applications using ZooKeeper demand high-performance from the service, and consequently, one important goal is the ability of having multiple outstanding client operations at a time. Zab enables multiple outstanding state changes by guaranteeing that at most one primary is able to broadcast state changes and have them incorporated into the state, and by using a synchronization phase while establishing a new primary. Before this synchronization phase completes, a new primary does not broadcast new state changes. Finally, Zab uses an identification scheme for state changes that enables a process to easily identify missing changes. This feature is key for efficient recovery.

Experiments and experience so far in production show that our design enables an implementation that meets the performance requirements of our applications. Our implementation of Zab can achieve tens of thousands of broadcasts per second, which is sufficient for demanding systems such as our Web-scale applications.

***Index Terms*—Fault tolerance, Distributed algorithms, Primary backup, Asynchronous consensus, Atomic broadcast**

I. Introduction

Atomic broadcast is a commonly used primitive in distributed computing and ZooKeeper is yet another application to use atomic broadcast. ZooKeeper is a highly-available coordination service used in production Web systems such as the Yahoo! crawler for over three years. Such applications often comprise a large number of processes and rely upon scheme [5], [6], [7] to maintain the state of replica processes consistent. With ZooKeeper, a primary process receives all incoming client requests, executes them, and propagates the resulting non-commutative, incremental state changes in the form of *transactions* to the backup replicas using *Zab*, the ZooKeeper atomic broadcast protocol. Upon primary crashes, processes execute a recovery protocol both to agree upon a common consistent state before resuming regular operation and to establish a new primary to broadcast state changes. To exercise the primary role, a process must have the support of a quorum of processes. As processes can crash and recover, there can be over time multiple primaries and in fact the same process may exercise the primary role multiple times. To distinguish the different primaries over time, we associate an instance value with each established primary. A given instance value maps to at most one process. Note that our notion of instance shares some of the properties of views of group communication [8], but it presents some key differences. With group communication, all processes in a given view are able to broadcast, and configuration changes happen when any process joins or leaves. With Zab, processes change to a new view (or primary instance) only when a primary crashes or loses support from a quorum.

Critical to the design of Zab is the observation that each state change is *incremental with respect to the previous state*, so there is an implicit dependence on the order of the state changes. State changes consequently cannot be applied in any arbitrary order, and it is critical to guarantee that a prefix of the state changes generated by a given primary are delivered and applied to the service state. State changes are idempotent and applying the same state change multiple times does not lead to inconsistencies as long as the application order is consistent

The Part-Time Parliament

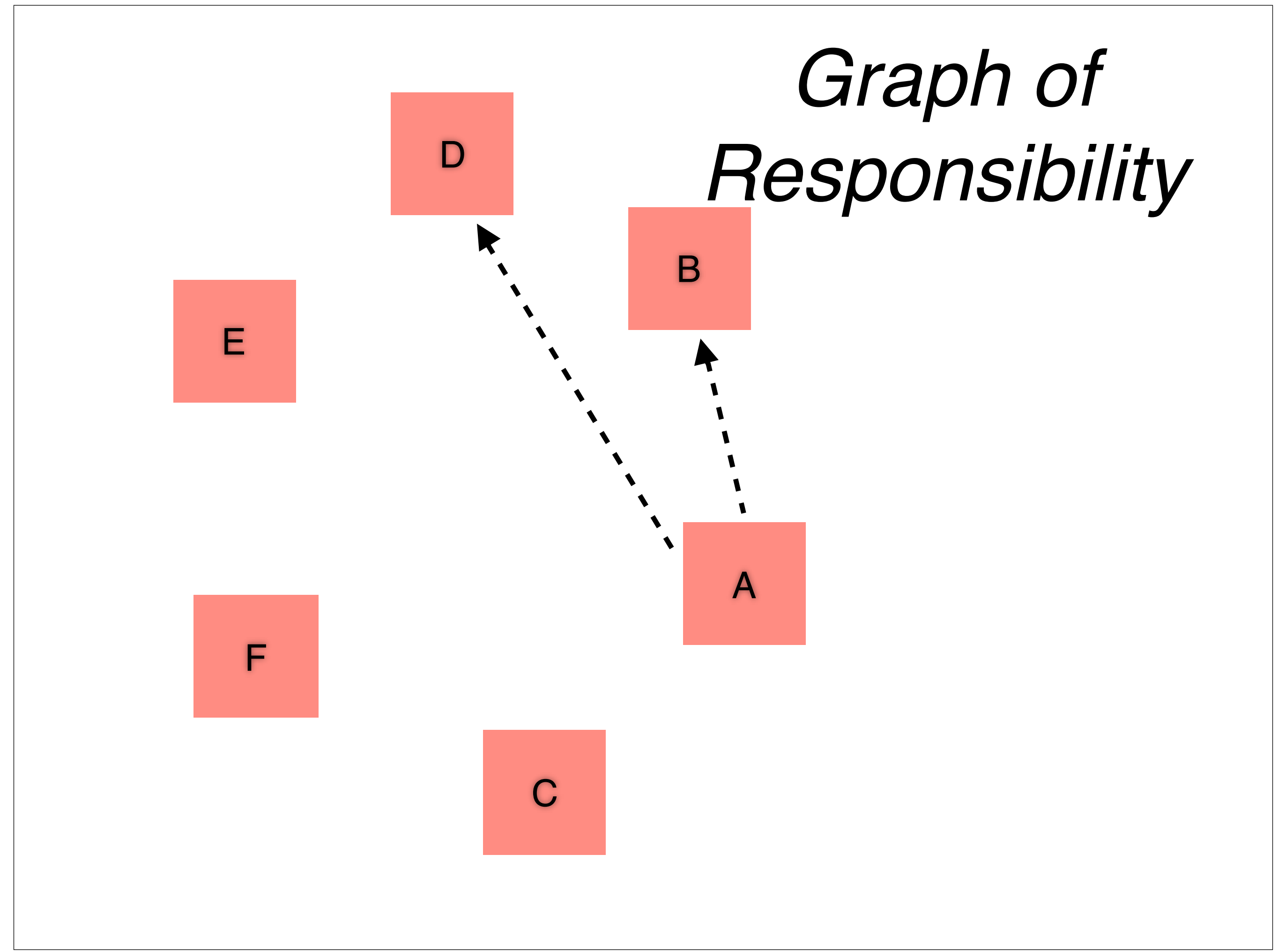
Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.

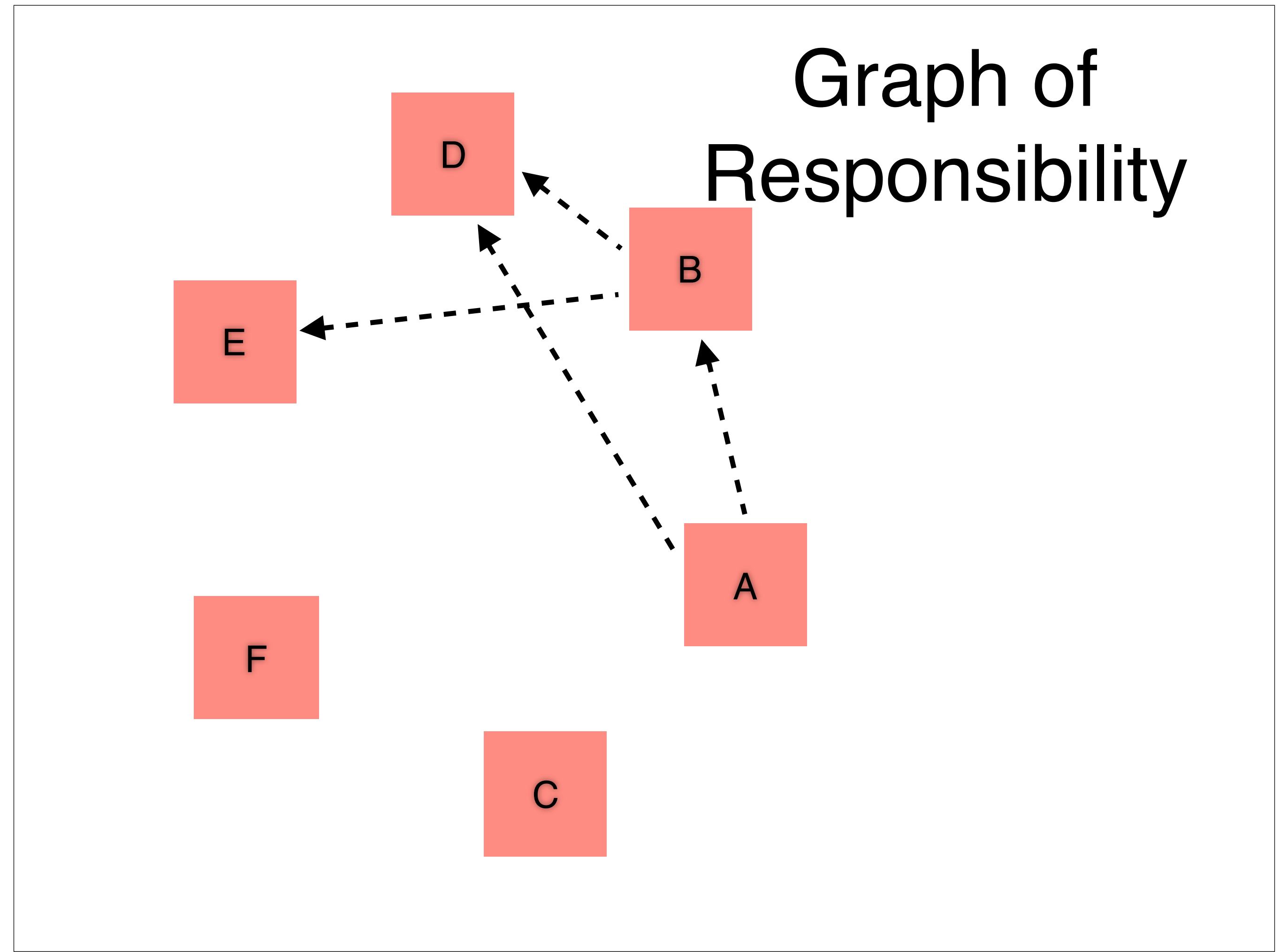
Strong Guarantees

Too Strong

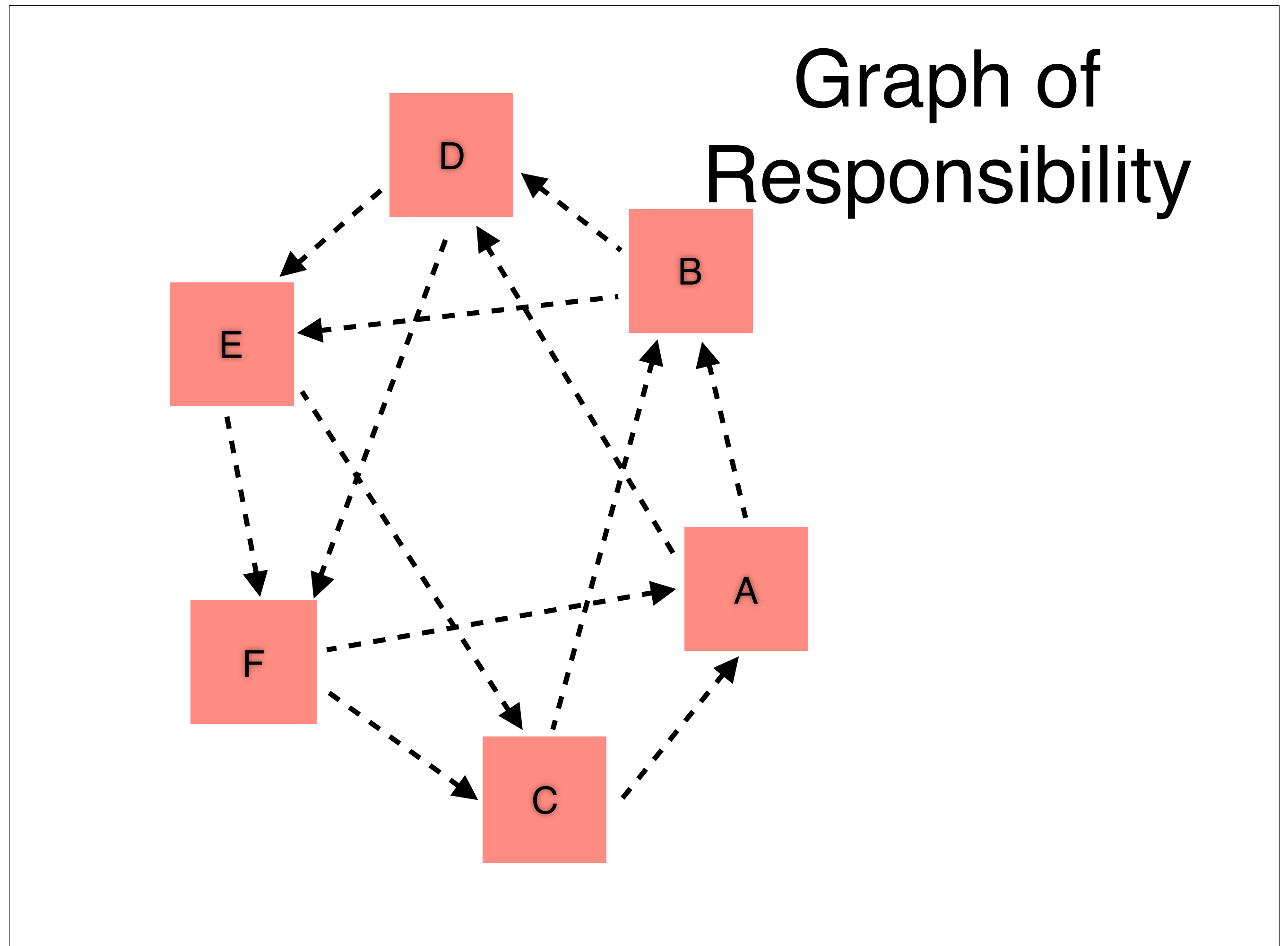
"Distributed systems, don't read the literature. Most of it is outdated and unimaginative. Invent and reinvent. The field is fertile. Really."



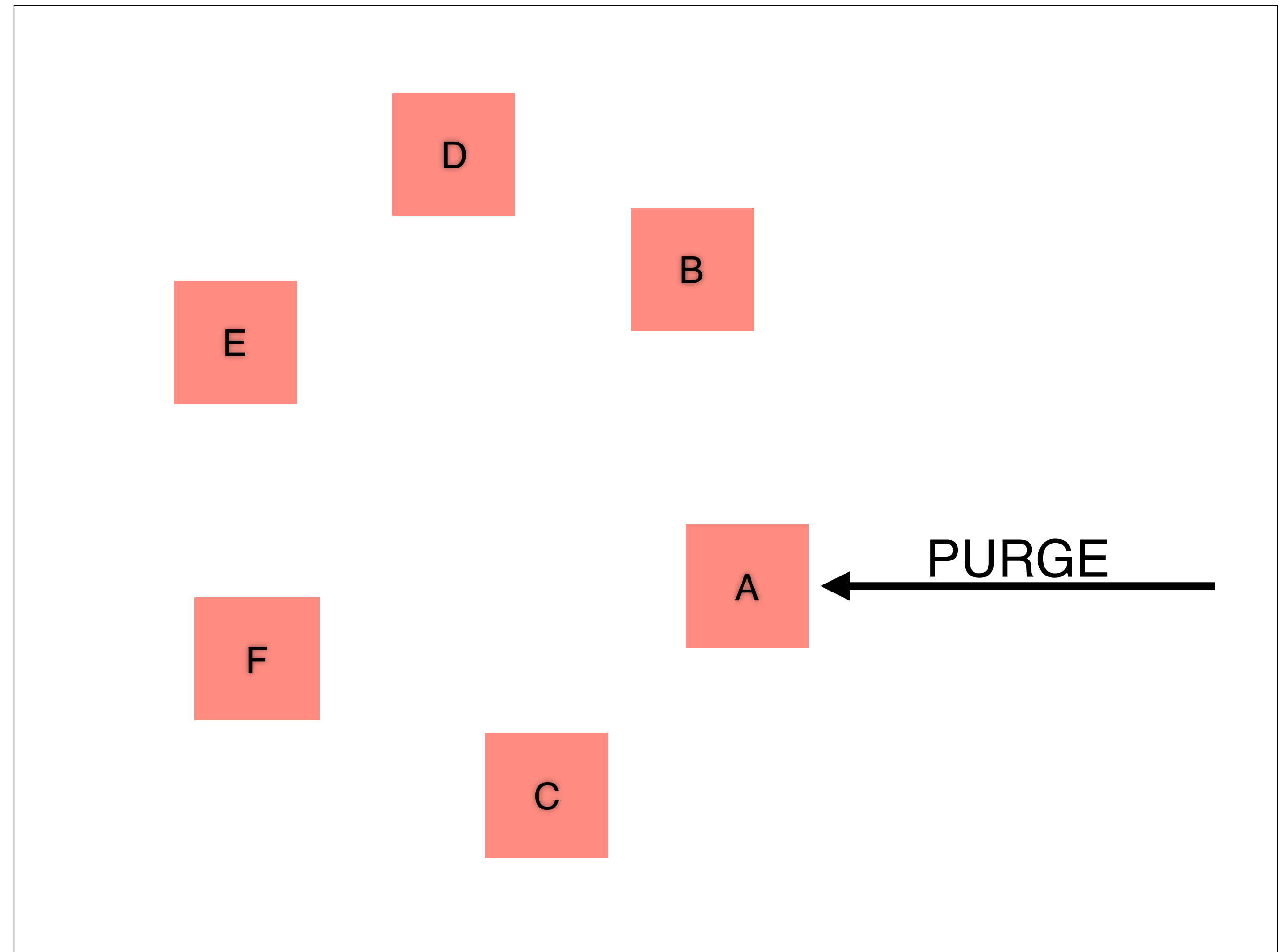
What we do is define a “graph of responsibility”. This defines which nodes are responsible for making sure each other stay up to date. So in this case, A is responsible for both B and D.



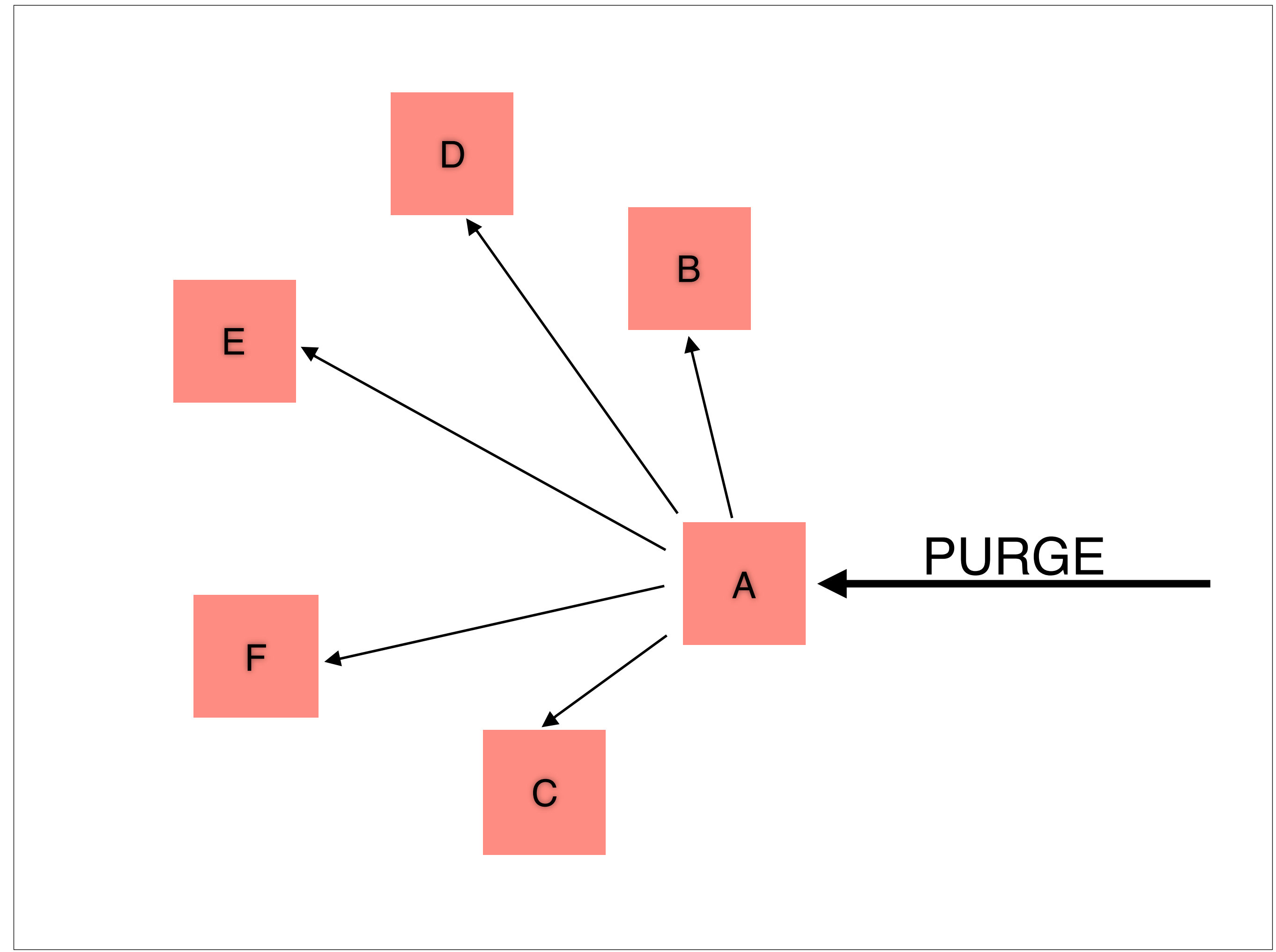
B is responsible for D and E.



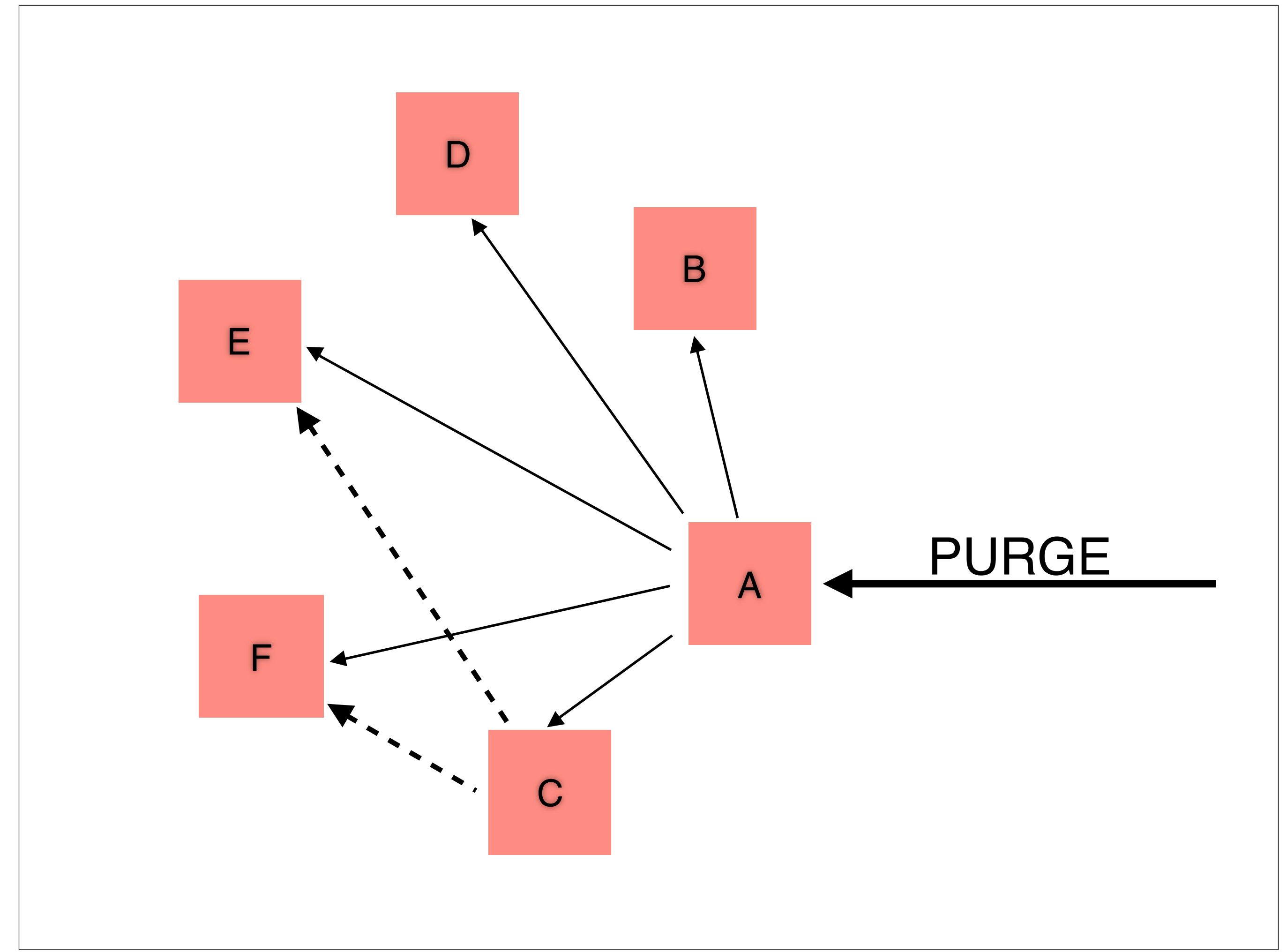
And so on...



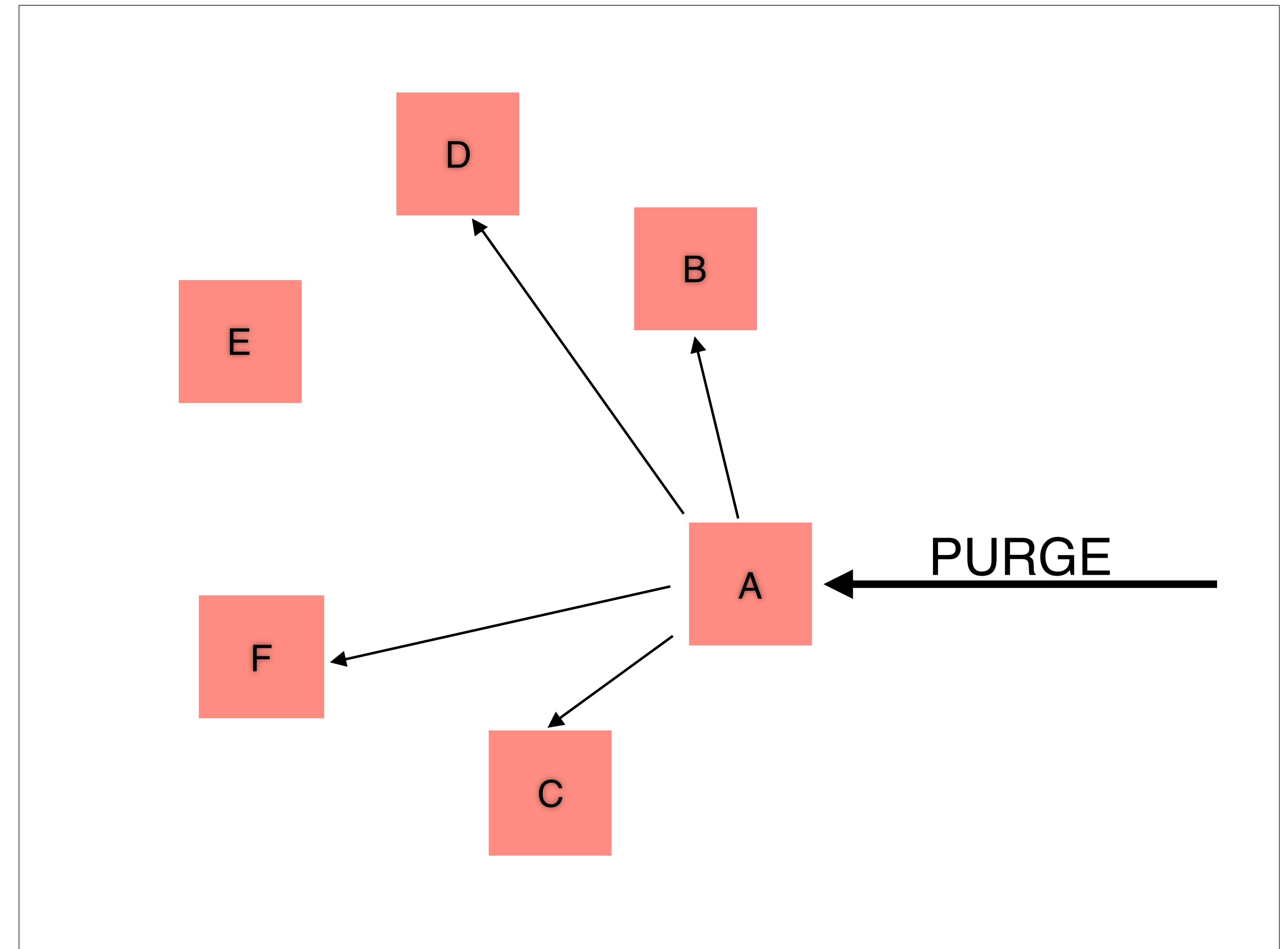
So, let’s follow a purge through this system. A purge request comes in to A.



A immediately forwards it via simple UDP messages to every other server.

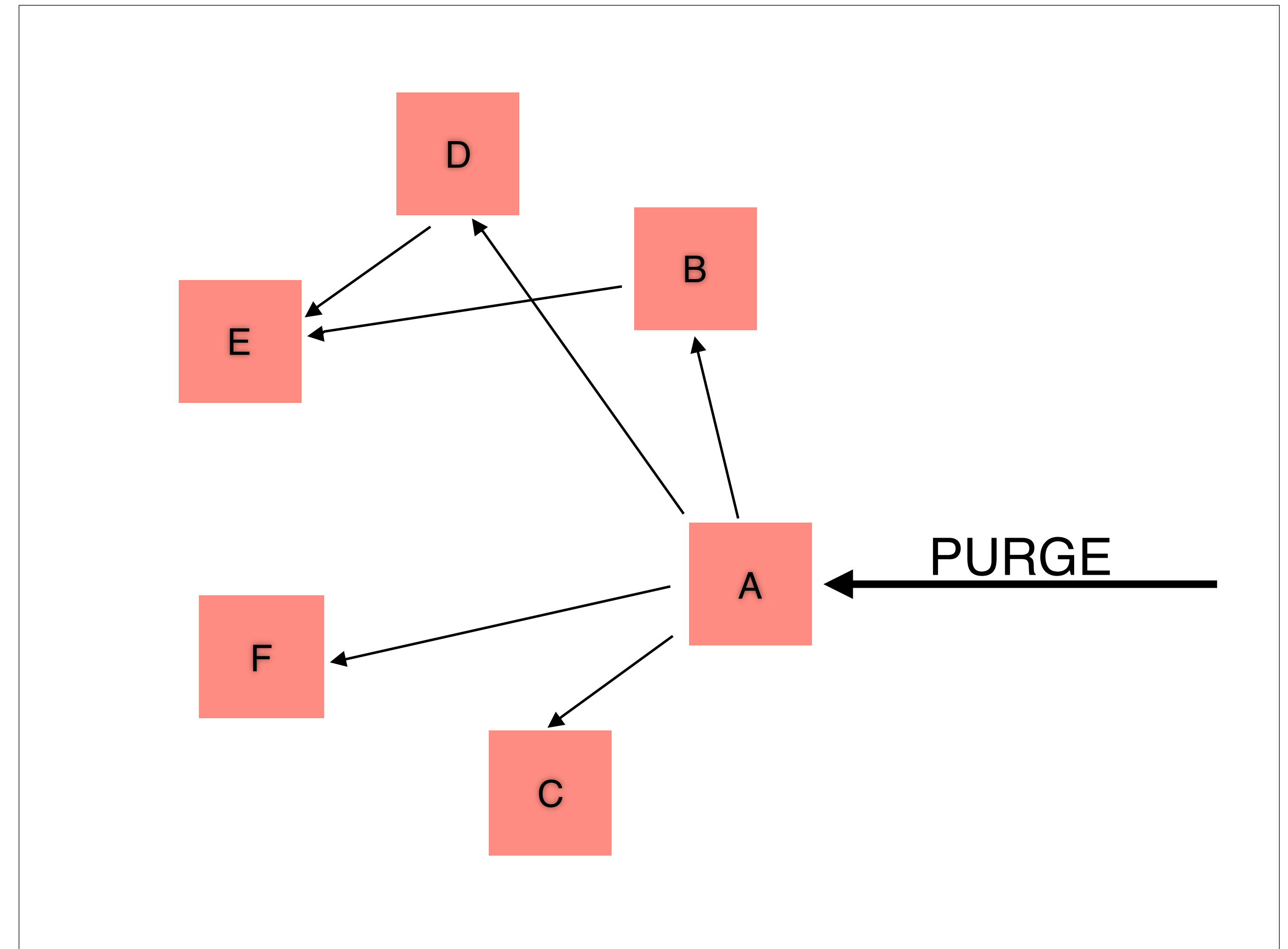


Each of the servers that receives a message then sends a “confirmation” to the server that is responsible for it.



What is more interesting is what happens when a message fails to reach a server.

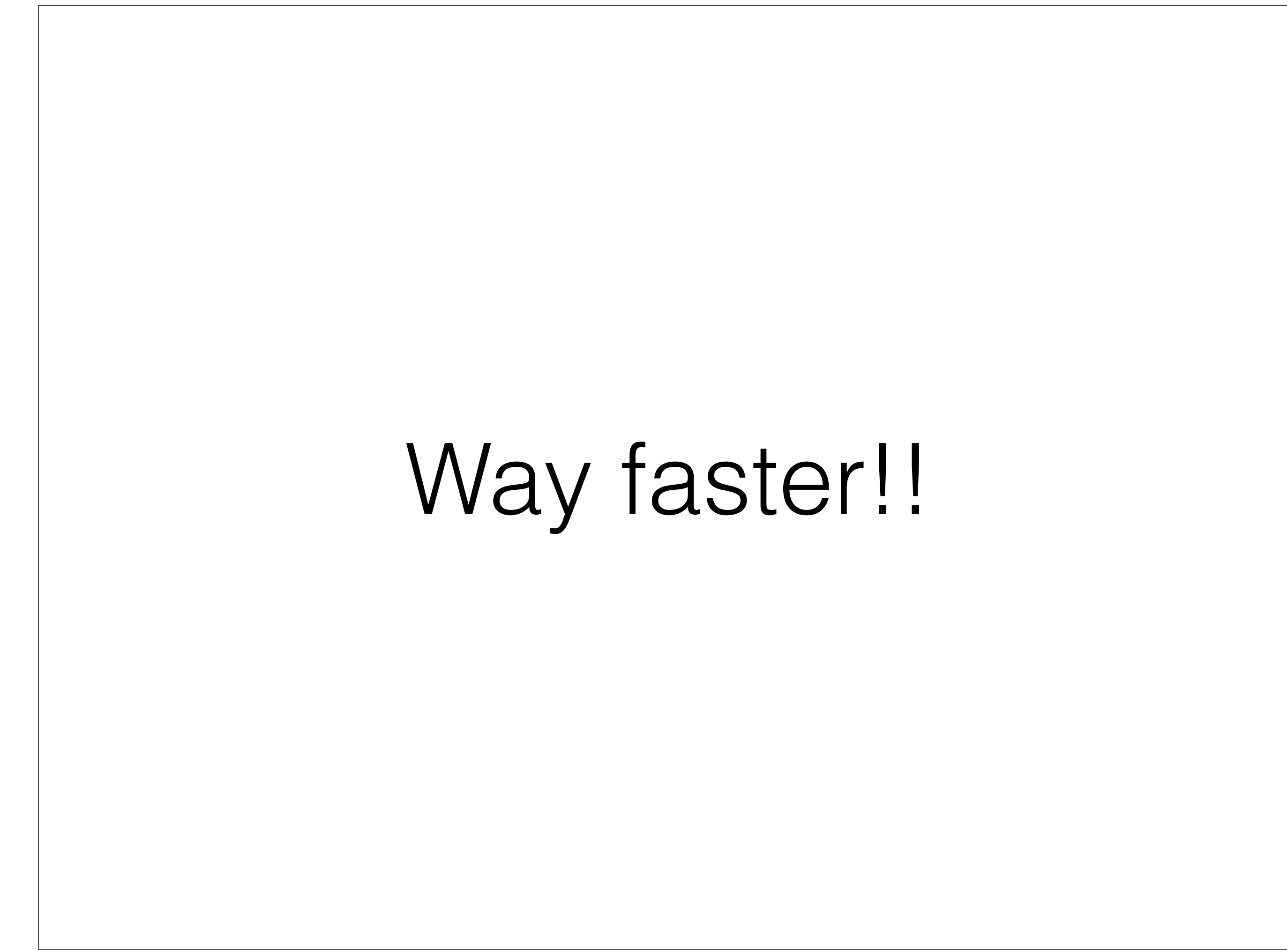
If a server receives a purge but does *not* get a confirmation from one of it’s “children”. It will send “reminders” to it.



So, in this case D and B will start sending reminders to E until it confirms receipt.

You can think of this as a primitive form of an “active anti-entropy”, which is a mechanism in which servers actively make sure that each other are up-to-date.

We ran a system designed this way for quite some time. And once again, it worked.



This system is much faster. It gets us close to the theoretical minimal latency in the happy path.

However, there are problems with it.

Arbitrary Partitions

The graph of responsibility must be designed very carefully to avoid having common network partitions cause the graph to become completely split. Additionally, even if it is carefully designed it can't handle *arbitrary* partitions. The best way to get close to fixing them is by increasing the number of nodes that are responsible for each other.

Which of course increases load on the system.

Unbounded Queues

Because every node is responsible for keeping other nodes up to date, it needs to know what each of its dependents have seen. Which means if a node is offline for a while, that queue grows arbitrarily large.

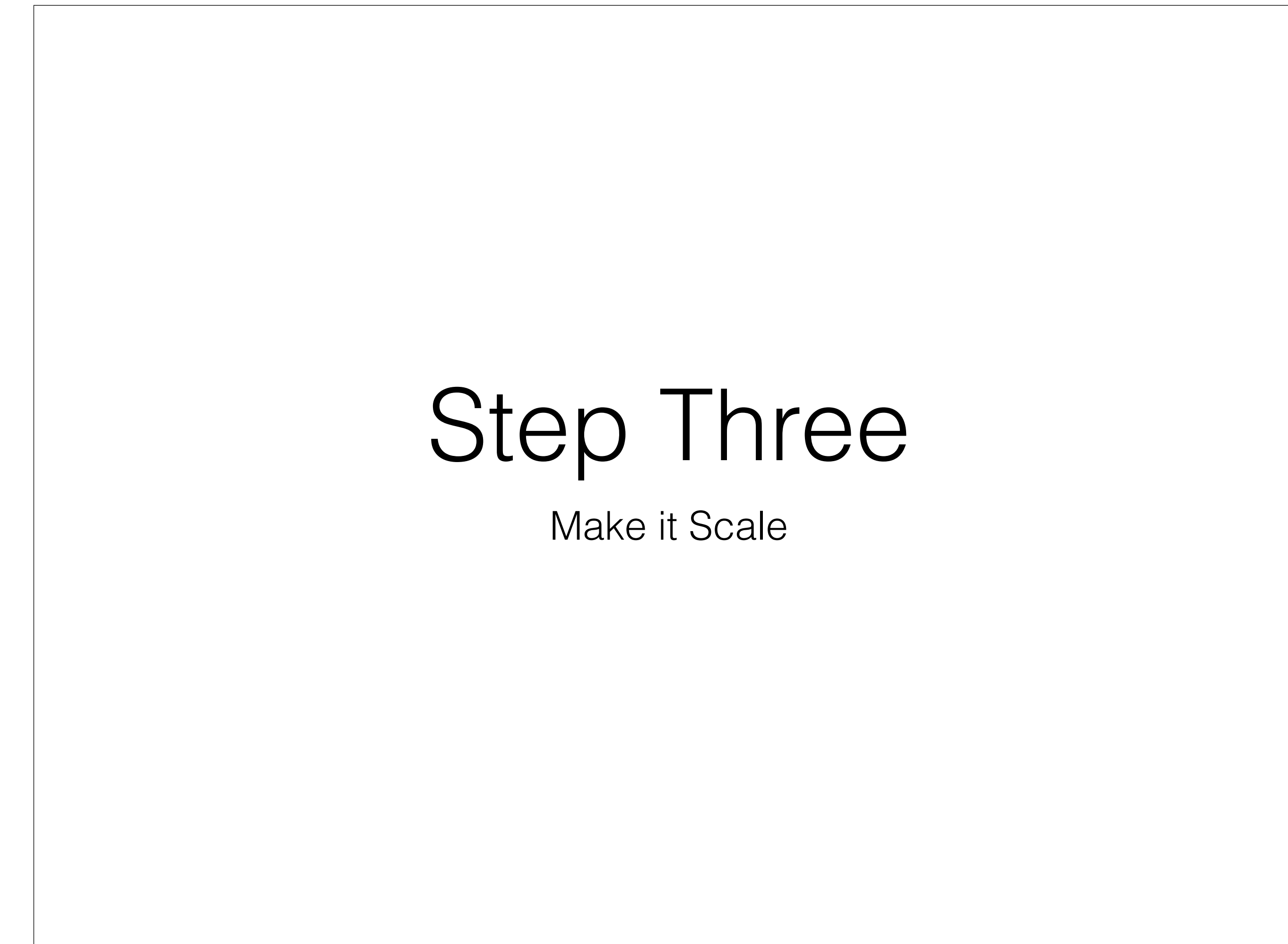
And the end result of that is Failure Dependence. One node failing means that multiple other nodes have to spend more time remembering messages and trying to send reminders to the failed node.

So, under duress this system is prone to having a single node failure become a multi-node failure, and a multi-node failure become a whole-system failure.

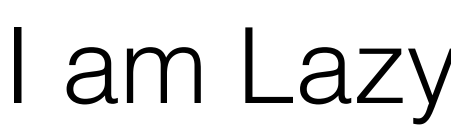
The problem with thinking real hard…

So, I said that we designed this problem by thinking really hard. The problem with that is that we didn't manage to find the existing research on this problem. It turns out that this type of system…

… was actually described in papers in the 1980s, when Devo was popular. The problems that we found with it are thus well-known. Luckily around that time, the venerable Bruce Spang started working with us.



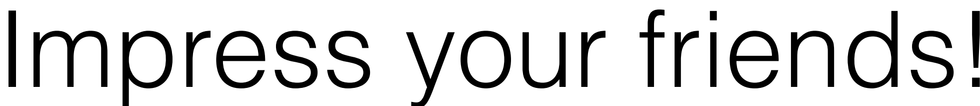
This is where I came in, and started working on building a system that scaled better and solved some of the problems with the previous one.



As Tyler showed just now, it turns out that inventing distributed algorithms is really hard. Even though Tyler came up with an awesome idea and implemented it well, it still had a bunch of problems that have been known since the eighties. I didn't want to think equally as hard, just to come up with something from five years later.

Read Papers

Instead, I decided to read papers and see if I could find something that we could use. Because we had a system in production that was working well enough, I had enough time to dig into the problem. But why would you read papers?



Papers are super cool and if you read them, you will also be cool.

Understand Problems

Get a better sense of the problem you are trying to solve, and learn about other ways people have tried to solve the same problem.

Learn what is impossible

Lots of papers prove that something is impossible, or show a bunch of problems with a system. By reading these papers, you can avoid a bunch of time trying to build a system that does something impossible and debugging it in production.

Find solutions to your problem

Finally, some papers may describe solutions to your problem. Not only will you be able to re-use the result from the paper, but you will also have a better chance of predicting how the thing will work in the future (since papers have graphs and shit). You may even find solutions to future problems along the way.

So I started reading papers by searching for maybe relevant things on google scholar.

Reliable Broadcast

The first class of papers that I came across attempted to solve the problem of reliable message broadcast. This is the problem of sending a message to a bunch of servers, and guaranteeing its delivery, which is a lot like our purging problem.

AN EFFICIENT RELIABLE BROADCAST PROTOCOL

M. Frans Kaashoek
Andrew S. Tanenbaum
Susan Flynn Hummel
Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Email: kaashoek@cs.vu.nl

ABSTRACT

Many distributed and parallel applications can make good use of broadcast communication. In this paper we present a (software) protocol that simulates reliable broadcast, even on an unreliable network. Using this protocol, application programs need not worry about lost messages. Recovery of communication failures is handled automatically and transparently by the protocol. In normal operation, our protocol is more efficient than previously published reliable broadcast protocols. An initial implementation of the protocol on 10 MC68020 CPUs connected by a 10 Mbit/sec Ethernet performs a reliable broadcast in 1.5 msec.

1. INTRODUCTION

Most current distributed operating systems are based on remote procedure call (RPC) [Birrell and Nelson 1984]. For many distributed and parallel applications, however, this sender-to-receiver-and-back communication style is inappropriate. What is frequently needed is *broadcasting*, in which an arbitrary one of the n user processes sends a message to the other $n - 1$ processes. Although broadcasting can always be simulated by sending $n - 1$ messages and waiting for the $n - 1$ acknowledgements, this algorithm is slow, inefficient, and wasteful of network bandwidth. In this paper we discuss a new protocol that allows 100% reliable

papers from the 80s like “an efficient reliable broadcast protocol”…

A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing

Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang
to appear in IEEE/ACM Transactions on Networking, December 1997

***Abstract*—This paper[1] describes SRM (Scalable Reliable Multicast), a reliable multicast framework for light-weight sessions and application level framing. The algorithms of this framework are efficient, robust, and scale well to both very large networks and very large sessions. The SRM framework has been prototyped in wb, a distributed whiteboard application, which has been used on a global scale with sessions ranging from a few to a few hundred participants. The paper describes the principles that have guided the SRM design, including the IP multicast group delivery model, an end-to-end, receiver-based model of reliability, and the application level framing protocol model. As with unicast communications, the performance of a reliable multicast delivery algorithm depends on the underlying topology and operational environment. We investigate that dependence via analysis and simulation, and demonstrate an adaptive algorithm that uses the results of previous loss recovery events to adapt the control parameters used for future loss recovery. With the adaptive algorithm, our reliable multicast delivery algorithm provides good performance over a wide range of underlying topologies.**

1 Introduction

Several researchers have proposed generic reliable multicast protocols, much as TCP is a generic transport protocol for reliable unicast transmission. In this paper we take a different view: unlike the unicast case where requirements for reliable, sequenced data delivery are fairly general, different multicast applications have widely different requirements for reliability. For example, some applications require that delivery obey a total ordering while many others do not. Some applications have many or all the members sending data while others have only one data source. Some applications have replicated data, for example in an n-redundant file store, so several members are capable of transmitting a data item while for others all data originates at a single source. These differences all affect the design of a reliable multicast protocol. Although one could design a protocol for the worst-case requirements, e.g., guaranteeing totally ordered

recognized. In 1990 Clark and Tennenhouse proposed a new protocol model called Application Level Framing (ALF) which explicitly includes an application's semantics in the design of that application's protocol [6]. ALF was later elaborated with a light-weight rendezvous mechanism based on the IP multicast distribution model, and with a notion of receiver-based adaptation for unreliable, real-time applications such as audio and video conferencing. The result, known as Light-Weight Sessions (LWS) [19], has been very successful in the design of wide-area, large-scale, conferencing applications. This paper further evolves the principles of ALF and LWS to add a framework for Scalable Reliable Multicast (SRM).

ALF says that the best way to meet diverse application requirements is to leave as much functionality and flexibility as possible to the application. Therefore SRM is designed to meet only the minimal definition of reliable multicast, i.e., eventual delivery of all the data to all the group members, without enforcing any particular delivery order. We believe that if the need arises, machinery to enforce a particular delivery order can be easily added on top of this reliable delivery service.

It has been argued [36, 34] that a single dynamically configurable protocol should be used to accommodate different application requirements. The ALF argument is even stronger: not only do different applications require different types of error recovery, flow control, and rate control mechanisms, but further, these mechanisms must explicitly account for the structure of the underlying application data itself.

SRM is also heavily based on the group delivery model that is the centerpiece of the IP multicast protocol [8]. In IP multicast, data sources simply send to the group's multicast address (a normal IP address chosen from a reserved range of addresses) without needing any advance knowledge of the group member-

…or “scalable reliable multicast”

Reliable Broadcast

As it turns out, these papers were a lot like the last version of the system. They tended to use retransmissions, with clever ways of building the retransmission graphs. This means that they had similar problems, so I kept looking for new papers by looking at other papers that cited these ones, and at other work by good authors.

Gossip Protocols

Eventually, I came across a class of protocols called gossip protocols that were written from the late 90s up until now

Epidemic Broadcast Trees *

João Leitão
University of Lisbon
jleitao@lasige.di.fc.ul.pt

José Pereira
University of Minho
jop@di.uminho.pt

Luís Rodrigues
University of Lisbon
ler@di.fc.ul.pt

Abstract

There is an inherent trade-off between epidemic and deterministic tree-based broadcast primitives. Tree-based approaches have a small message complexity in steady-state but are very fragile in the presence of faults. Gossip, or epidemic, protocols have a higher message complexity but also offer much higher resilience.

This paper proposes an integrated broadcast scheme that combines both approaches. We use a low cost scheme to build and maintain broadcast trees embedded on a gossip-based overlay. The protocol sends the message payload preferably via tree branches but uses the remaining links of the gossip overlay for fast recovery and expedite tree healing. Experimental evaluation presented in the paper shows that our new strategy has a low overhead and that is able to support large number of faults while maintaining a high reliability.

1. Introduction

Many systems require highly scalable and reliable broadcast primitives. These primitives aim at ensuring that all correct participants receive all broadcast messages, even in the presence of network omissions or node failures.

papers like plumtree

Sprinkler — Reliable Broadcast for Geographically Dispersed Datacenters

Haoyan Geng and Robbert van Renesse

Cornell University, Ithaca, New York, USA

Abstract. This paper describes and evaluates Sprinkler, a reliable high-throughput broadcast facility for geographically dispersed datacenters. For scaling cloud services, datacenters use caching throughout their infrastructure. Sprinkler can be used to broadcast update events that invalidate cache entries. The number of recipients can scale to many thousands in such scenarios. The Sprinkler infrastructure consists of two layers: one layer to disseminate events among datacenters, and a second layer to disseminate events among machines within a datacenter. A novel garbage collection interface is introduced to save storage space and network bandwidth. The first layer is evaluated using an implementation deployed on Emulab. For the second layer, involving thousands of nodes, we use a discrete event simulation. The effect of garbage collection is analyzed using simulation. The evaluation shows that Sprinkler can disseminate millions of events per second throughout a large cloud infrastructure, and garbage collection is effective in workloads like cache invalidation.

Keywords: Broadcast, performance, fault tolerance, garbage collection

1 Introduction

Today's large scale web applications such as Facebook, Amazon, eBay, Google+, and so on, rely heavily on caching for providing low latency responses to client queries. Enterprise data is stored in reliable but slow back-end databases. In order to be able to keep up with load and provide low latency responses, client query results are computed and opportunistically cached in memory on many thousands of machines throughout the organization's various datacenters [21]. But when a database is updated, all affected cache entries have to be invalidated. Until this is completed, inconsistent data can be exposed to clients. Since the databases cannot keep track of where these cache entries are, it is necessary to multicast an invalidation notification to all machines that may have cached query results. The rate of such invalidations can reach hundreds of thousands per second. If any invalidation gets lost, inconsistencies exposed to clients may be

or sprinkler

the main difference between these papers and reliable broadcast papers was that they were designed to be much more scalable

- tens of thousands of servers
- hundreds of thousands or millions of messages per second

to get this higher scale, usually these systems provide probabilistic guarantees about whether a message will be delivered, instead of guaranteeing that all messages will always be delivered.

Bimodal Multicast

KENNETH P. BIRMAN
Cornell University
MARK HAYDEN
Digital Equipment Corporation/Compaq
OZNUR OZKASAP and ZHEN XIAO
Cornell University
MIHAI BUDIU
Carnegie Mellon University
and
YARON MINSKY
Cornell University

There are many methods for making a multicast protocol "reliable." At one end of the spectrum, a reliable multicast protocol might offer atomicity guarantees, such as all-or-nothing delivery, delivery ordering, and perhaps additional properties such as virtually synchronous addressing. At the other are protocols that use local repair to overcome transient packet loss in the network, offering "best effort" reliability. Yet none of this prior work has treated stability of multicast delivery as a basic reliability property, such as might be needed in an internet radio, television, or conferencing application. This article looks at reliability with a new goal: development of a multicast protocol which is reliable in a sense that can be rigorously quantified and includes throughput stability guarantees. We characterize this new protocol as a "bimodal multicast" in reference to its reliability model, which corresponds to a family of bimodal probability distributions. Here, we introduce the protocol, provide a theoretical analysis of its behavior, review experimental results, and discuss some candidate applications. These confirm that bimodal multicast is reliable, scalable, and that the protocol provides remarkably stable delivery throughput.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network

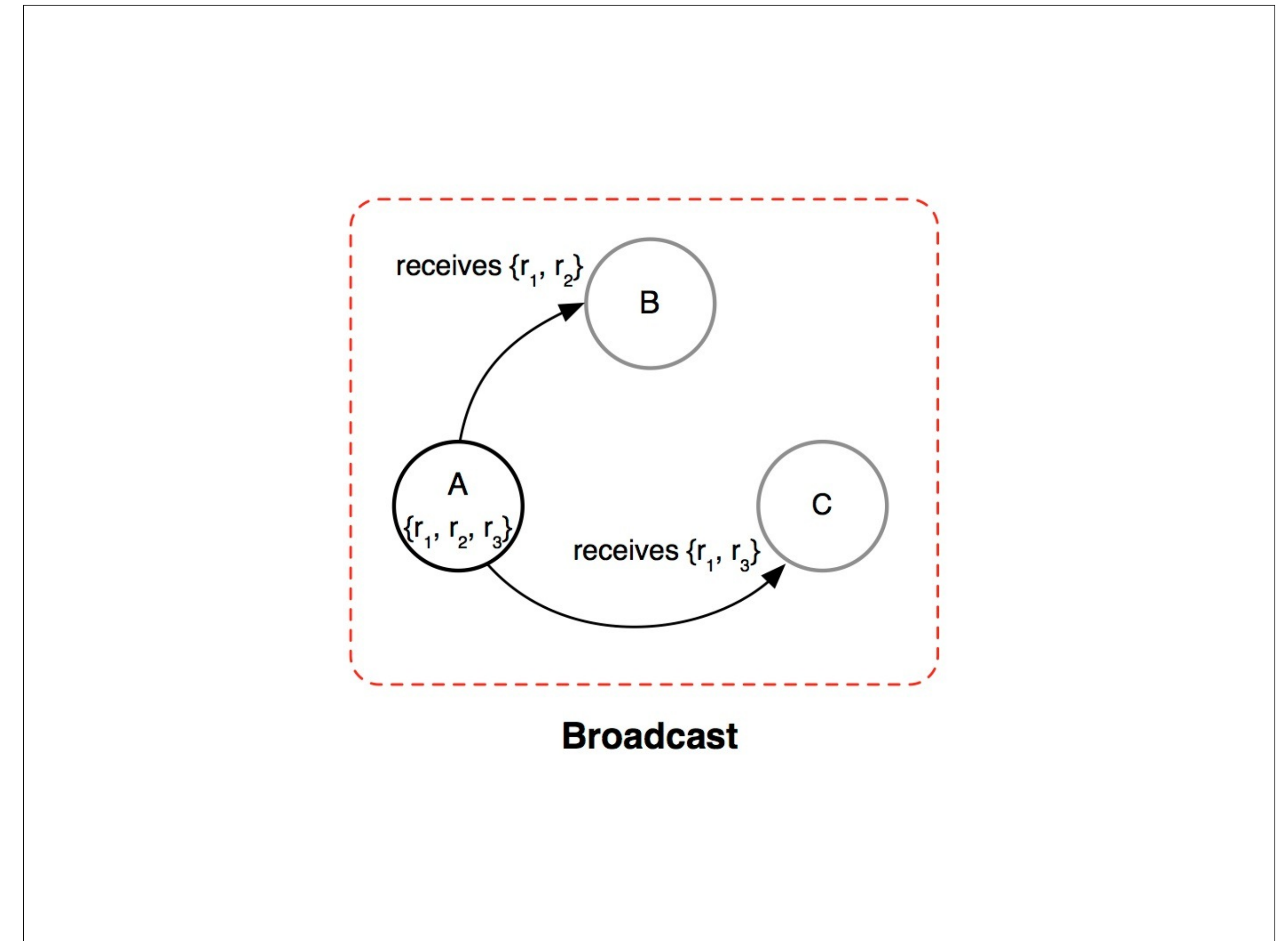
This work was supported by DARPA/ONR contracts N0014-96-1-10014 and ARPA/RADC F30602-96-1-0317, the Cornell Theory Center, and the Turkish Research Foundation.
Authors' addresses: K. P. Birman, Department of Computer Science, Cornell University, 4126 Upson Hall, Ithaca, NY 14853; email: ken@cs.cornell.edu; M. Hayden, Systems Research Center, Digital Equipment Corporation/Compaq, 130 Lytton Avenue, Palo Alto, CA 94301; email: hayden@src.dec.com; O. Ozkasap and Z. Xiao, Department of Computer Science, Cornell University, 4126 Upson Hall, Ithaca, NY 14853; email: ozkasap@cs.cornell.edu; xiao@cs.cornell.edu; M. Budiu, Department of Computer Science, Carnegie Mellon University, Ithaca, NY 14853; email: mihaib@cs.cmu.edu; Y. Minsky, Department of Computer Science, Cornell University, 4126 Upson Hall, Ithaca, NY 14853.
Permission to make digital/hard copy of part or all of this work for personal or classroom use

after reading a bunch of papers, we eventually decided to implement bimodal multicast

Bimodal Multicast

- Quickly broadcast message to all servers
- Gossip to recover lost messages

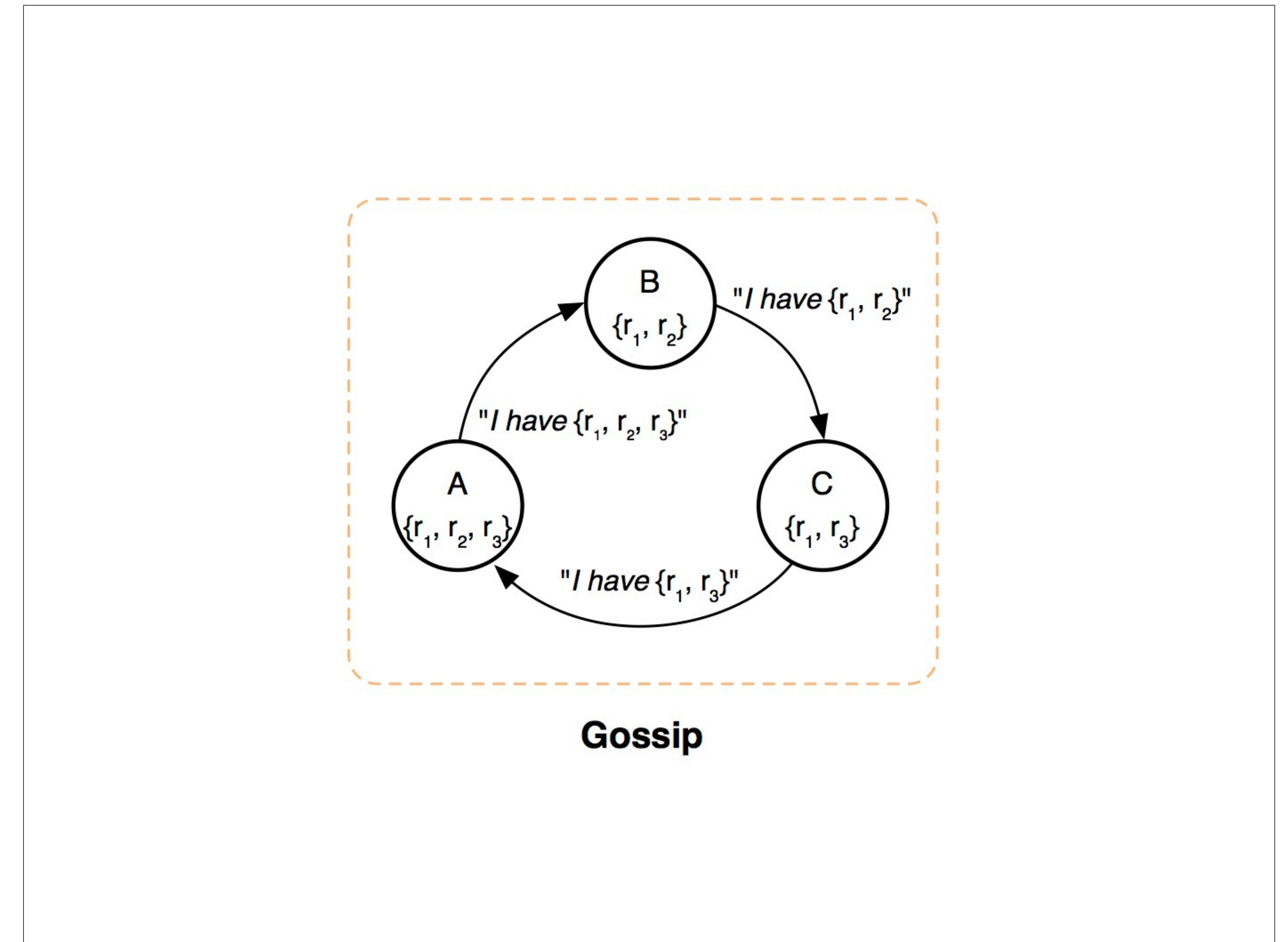
two phases: broadcast and gossip



send message to all other servers as quickly as possible

it doesn’t matter if it’s actually delivered here

you can use ip multicast if it’s available, udp in a for loop like us, a carrier pigeon, whatever…

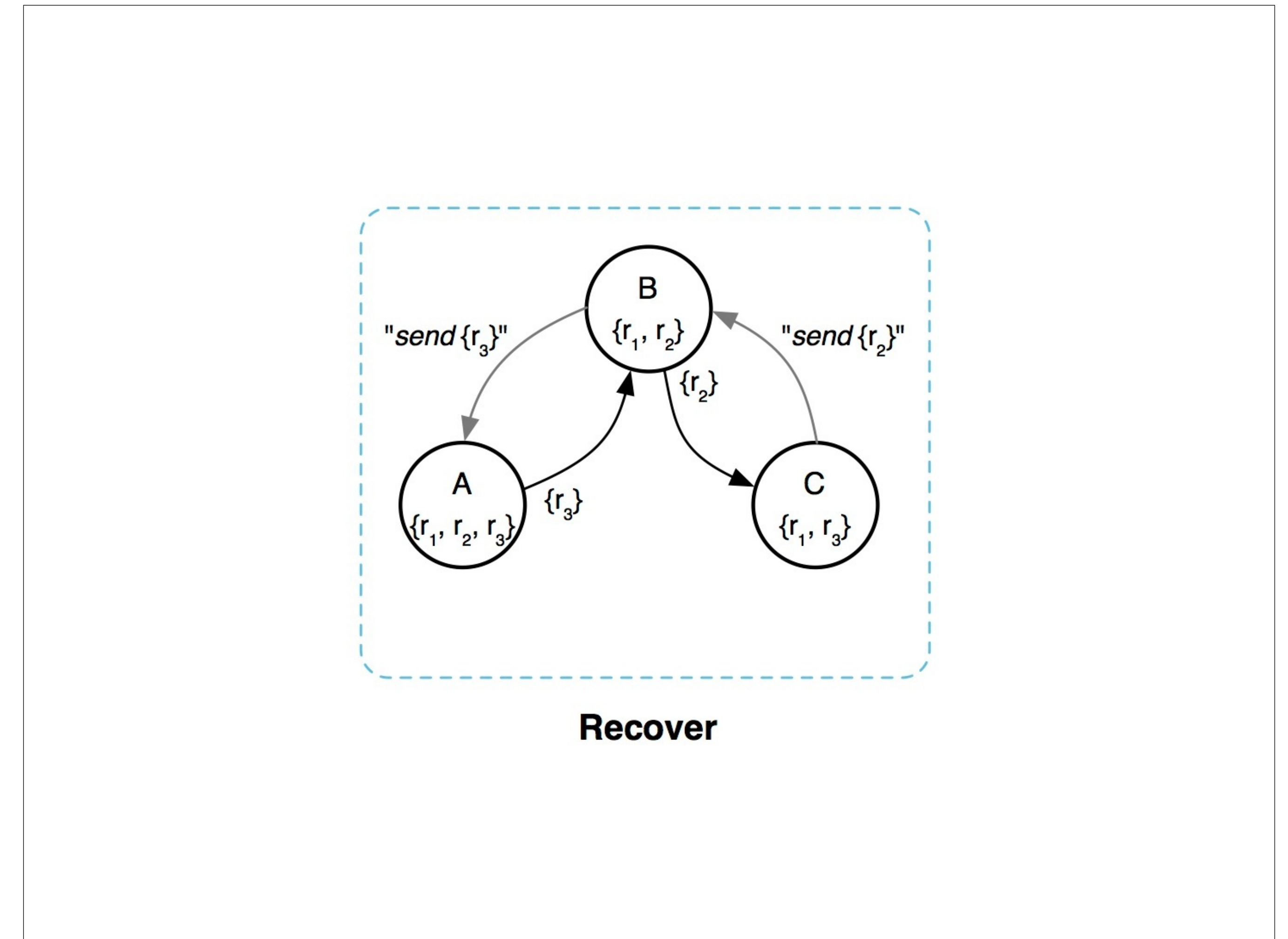


every server picks another server at random and sends a digest of all the messages they know about

- a picks b, b picks c, …

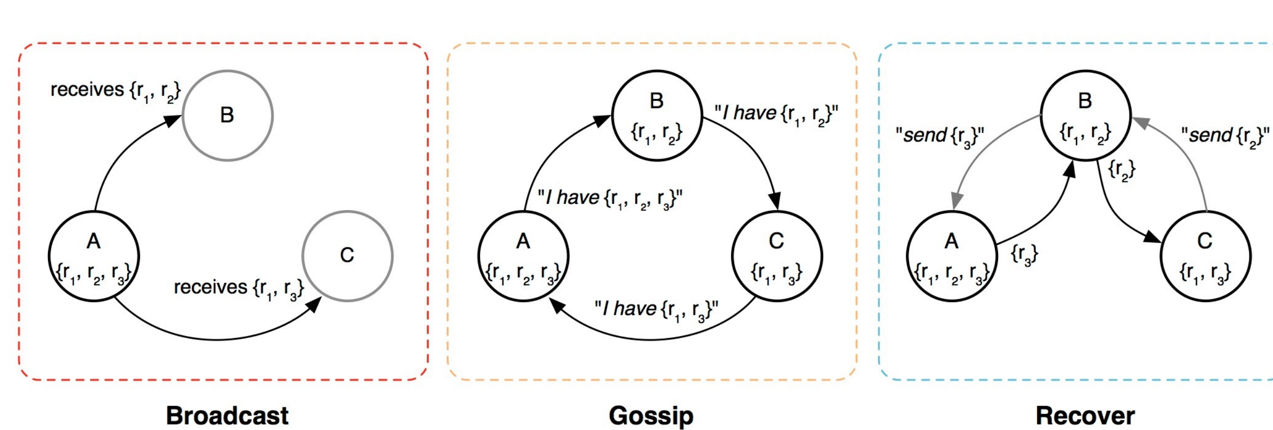
a server looks at the digest it received, and checks if it has any messages missing

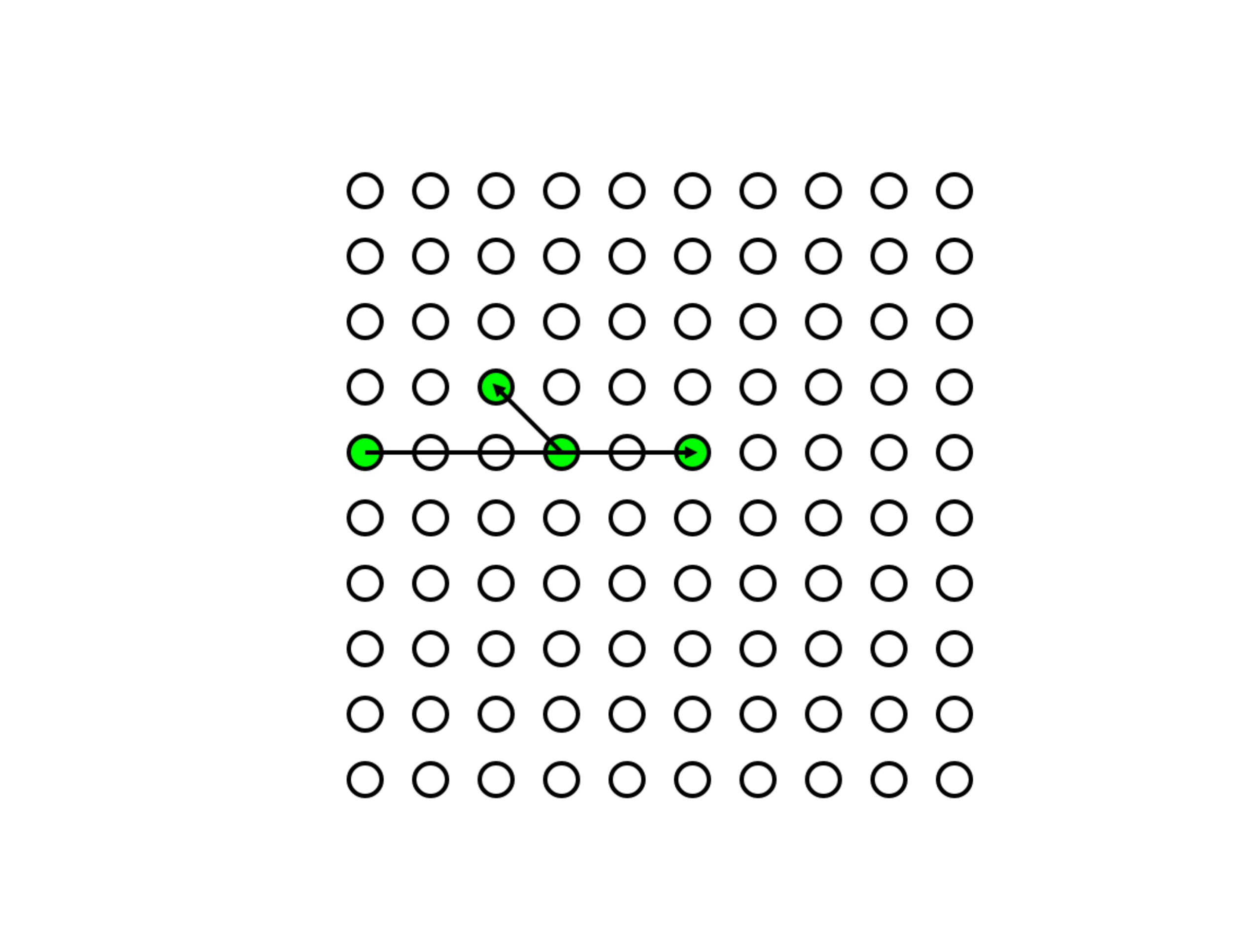
- b is missing 3, c is missing 2



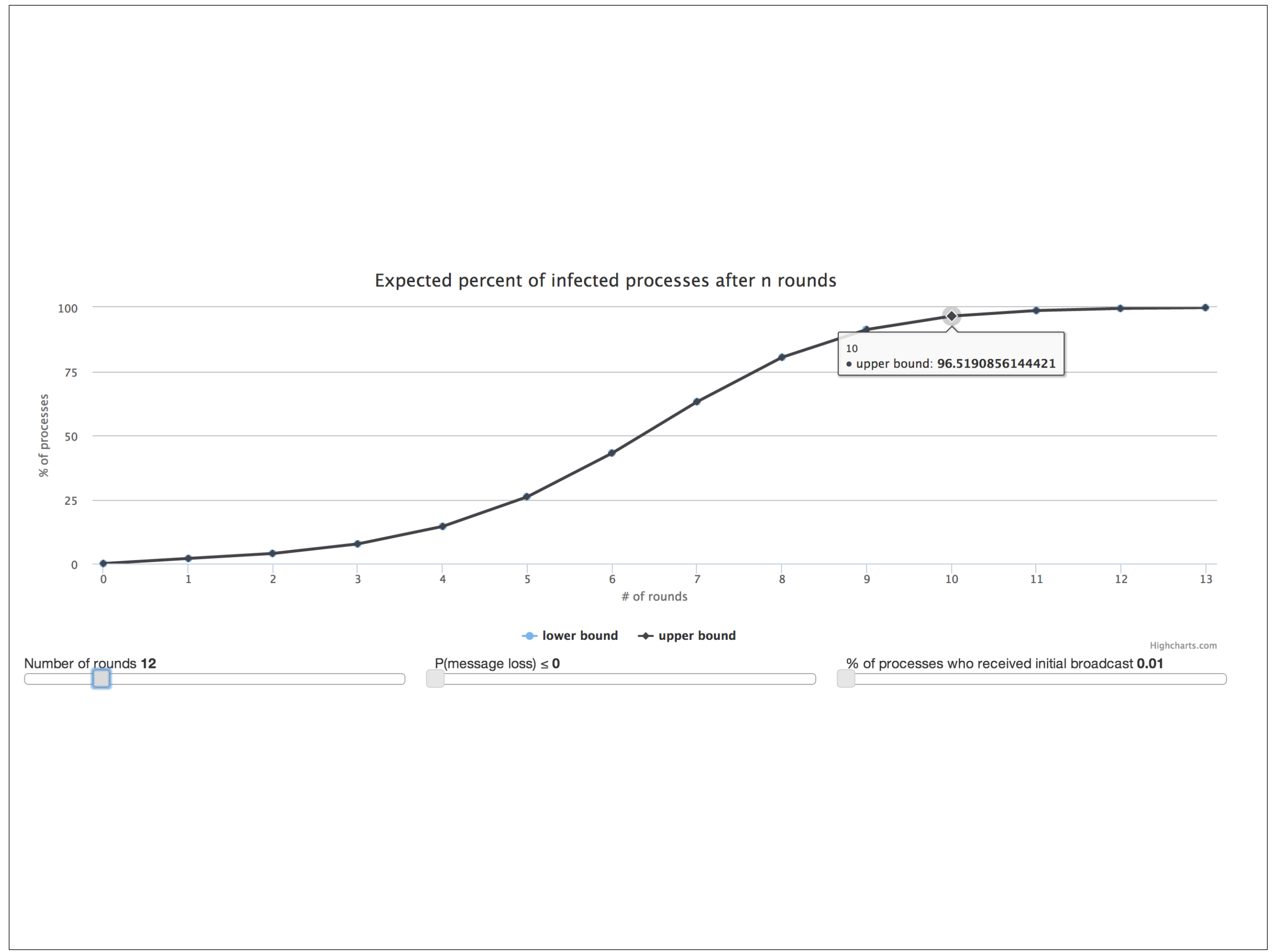
each server asks for any missing messages to be resent

Questions?





after reading the paper, we wanted more intuition about how this algorithm would actually work on many servers. we decided to implement a small simulation to figure it out.



- we still wanted a better guarantee before deploying it into production.
- the paper includes a bunch of math to predict the expected % of servers receiving a message after some number of round of gossip
- describe graph
- after 10 rounds, 97% of servers have message.
- turns out to be independent of the number of servers
- good enough for us

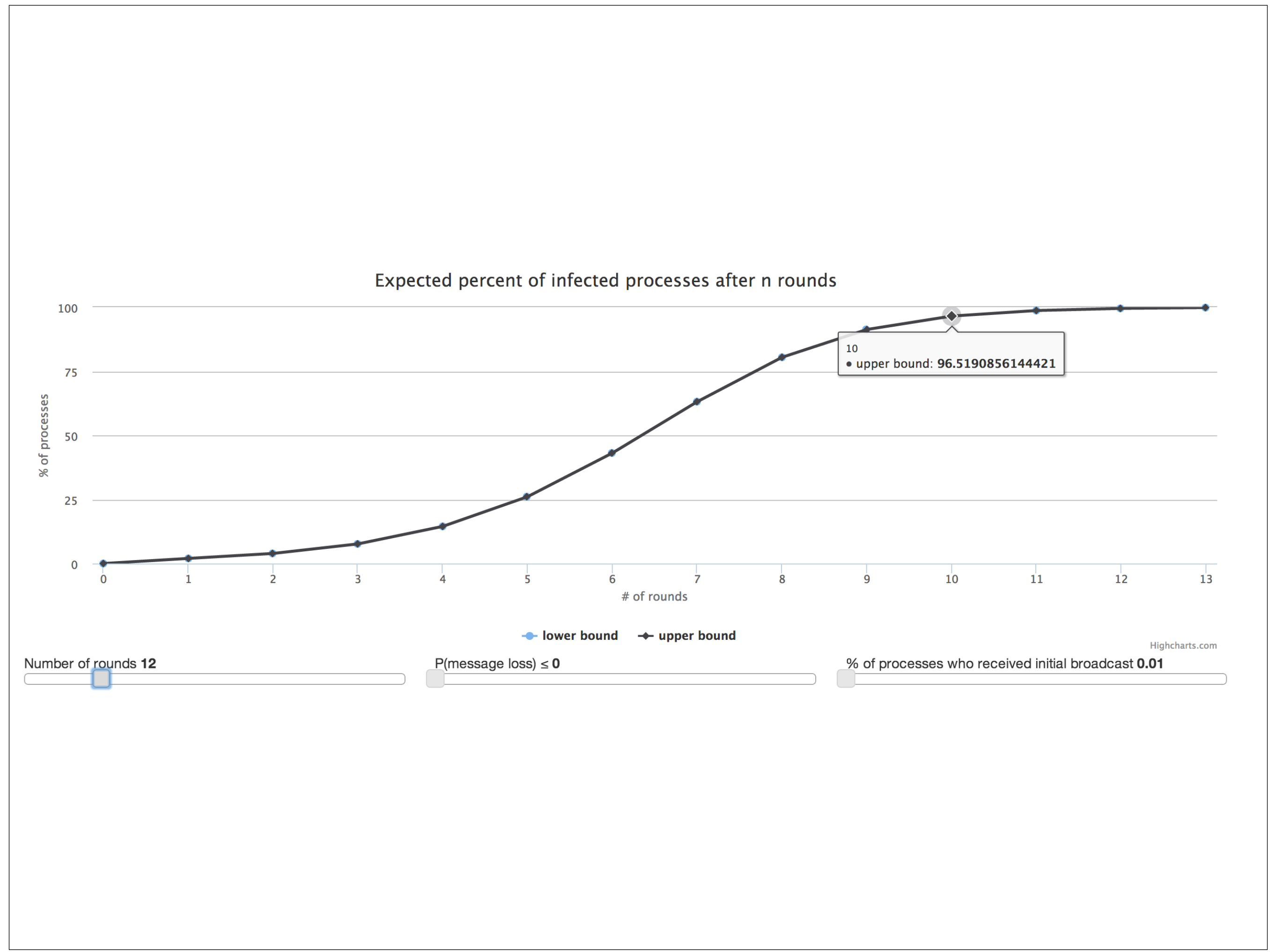
One Problem

Computers have limited space

started to implement it, ran across this problem

it needs to keep enough messages to recover for another server

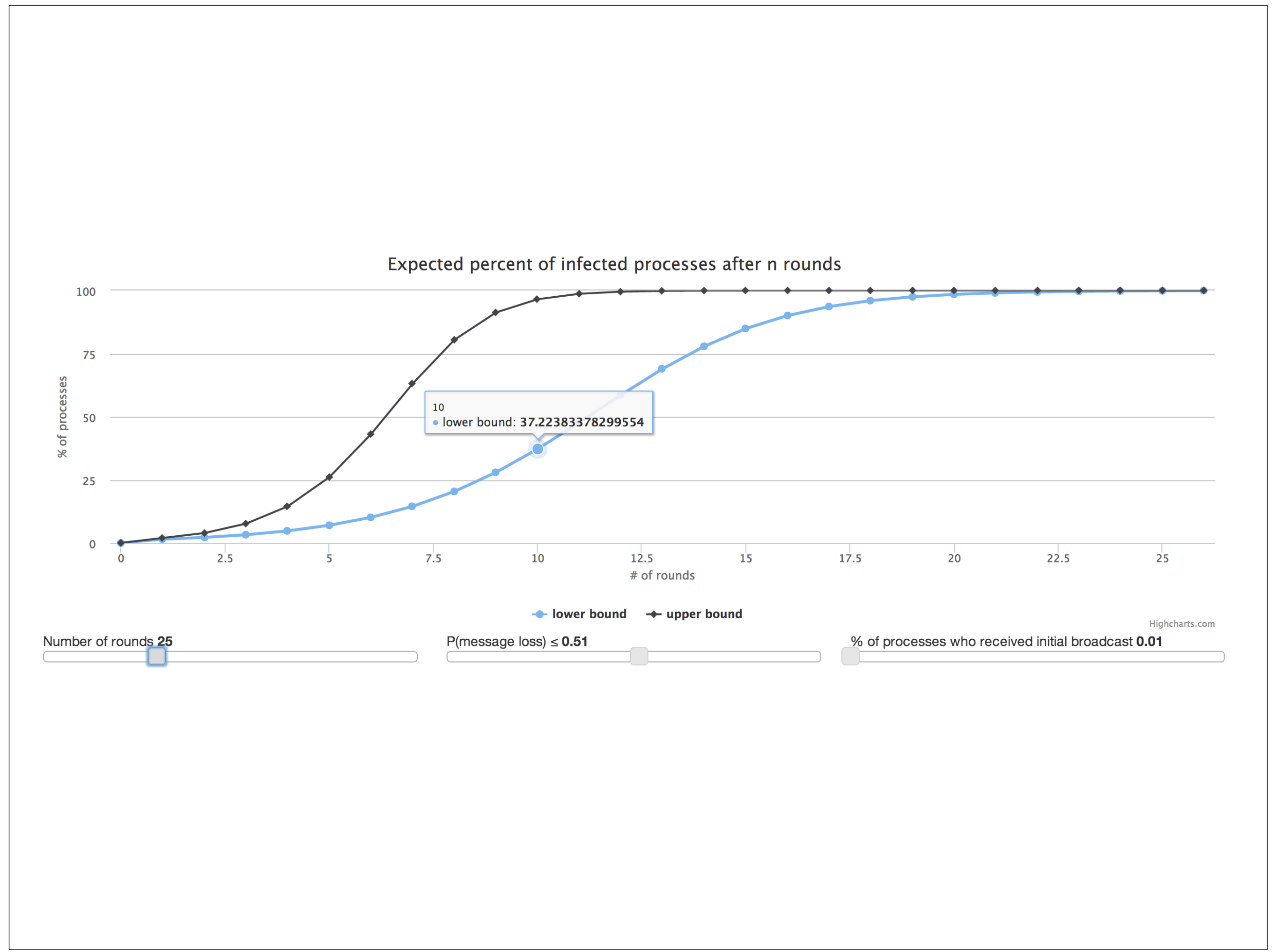
throw away messages to bound resource usage



- paper throws messages away after 10 rounds (97%)
- this makes sense during normal operation where there is low packet loss
- however, we often see more packet loss. we don't deal with theory, we deal with real computers…

Computers are Terrible

We see high packet loss *all the time*



- same graph as before, this time with 50% packet loss
- 40% of servers isn't good enough
- we'll probably lose purges during network outages, get calls from customers, etc…

The Digest

"I have 1, 2, 3, …"

why would the paper throw away after 10 rounds?

digest is a list, which is limited by bandwidth

need to limit the size of the digest

The Digest

Doesn't Have to be a List

it can be any data structure we want, as long as another node can understand it.

The Digest

Send ranges of ids of known messages

“messages 1 to 3 and 5 to 1,000,000"

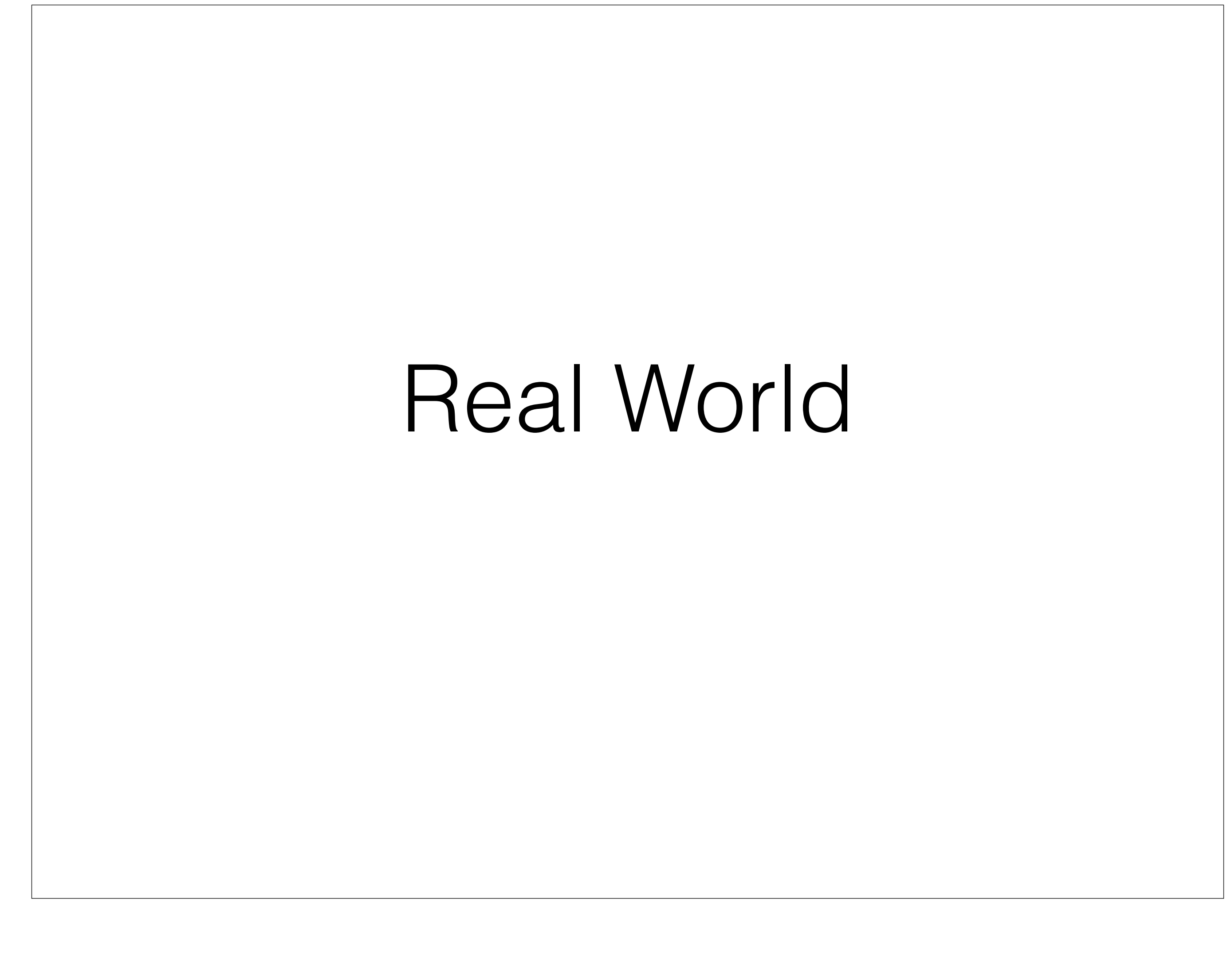
- normally just a few integers to represent millions of messages
- we keep messages around for a day, or about 80k rounds

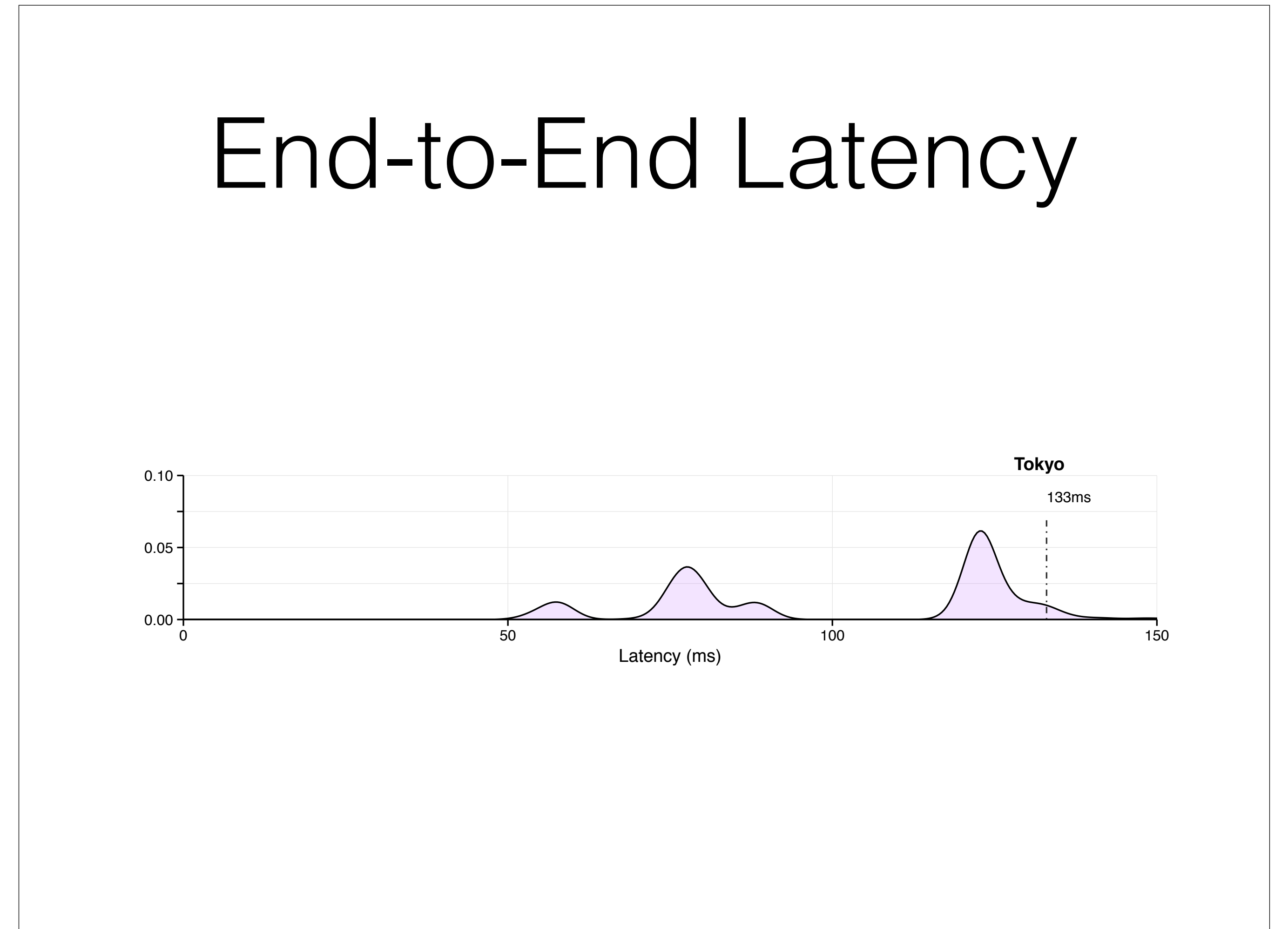
same graph, 80k rounds, 99% packet loss

99.9999999999% expected percent of servers to receive message

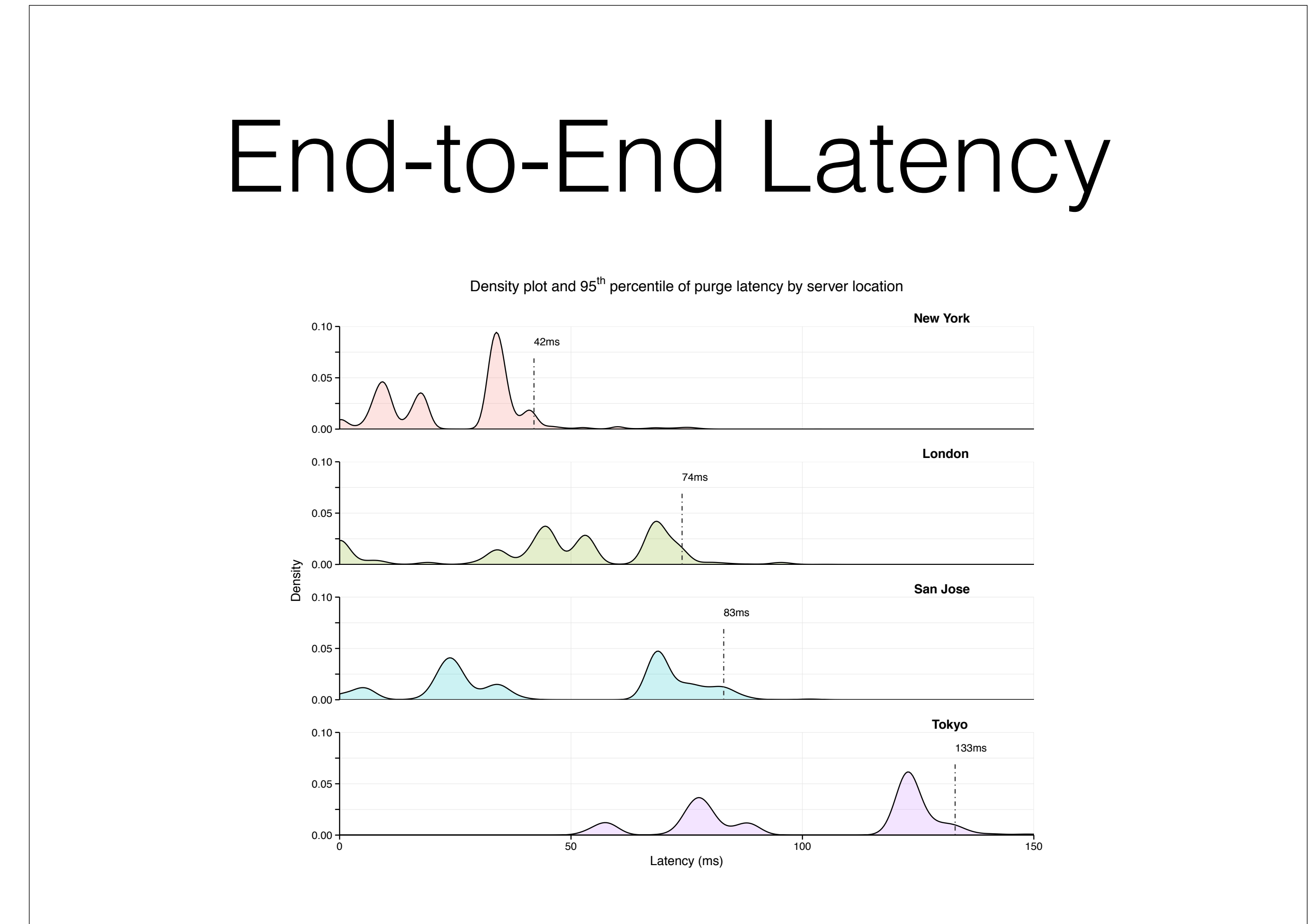
this is cool

as long as you know what that probability is


Real World

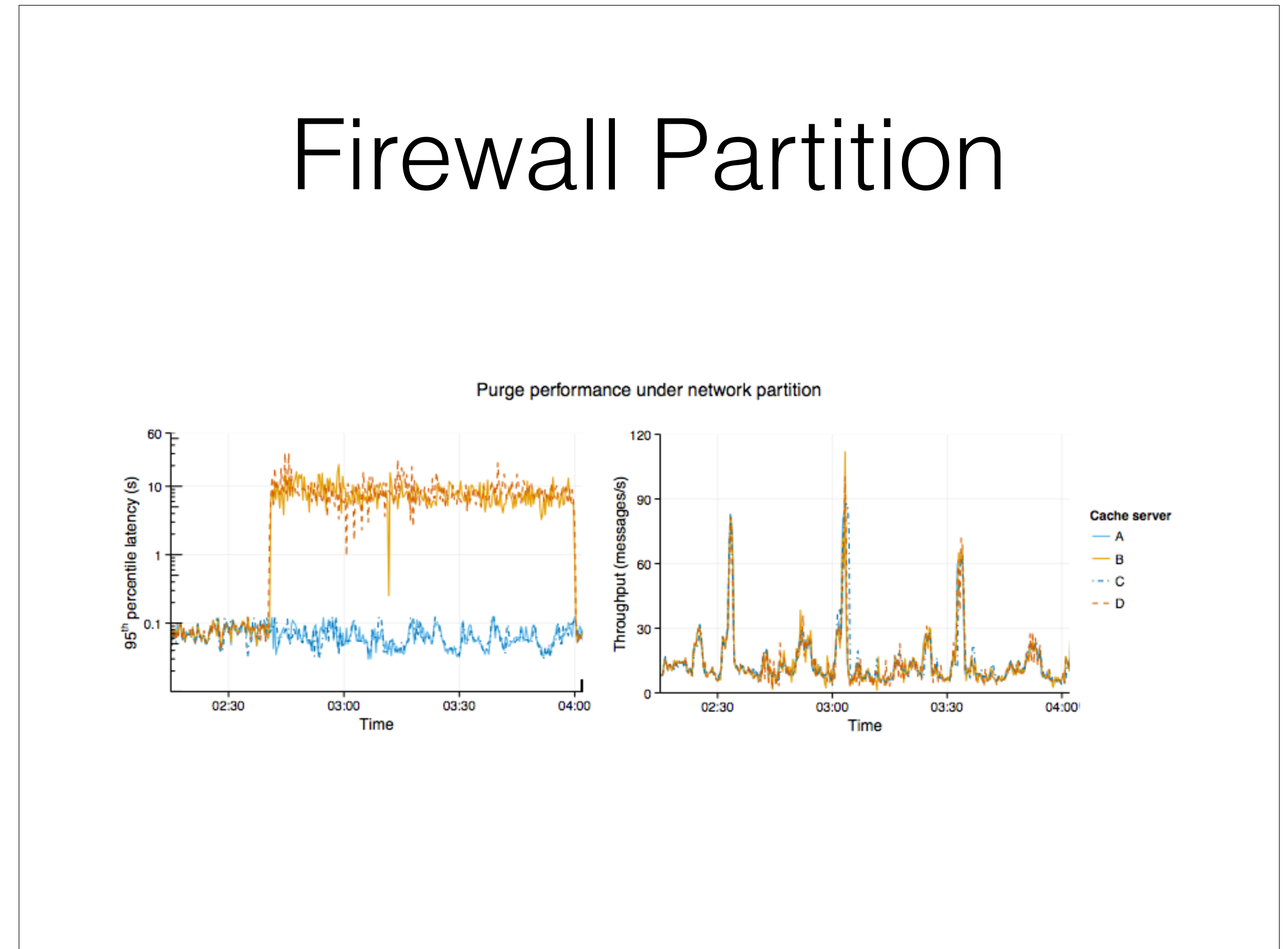


- usually < 0.1% packet loss on a link
- 95th percentile delivery latency is network latency

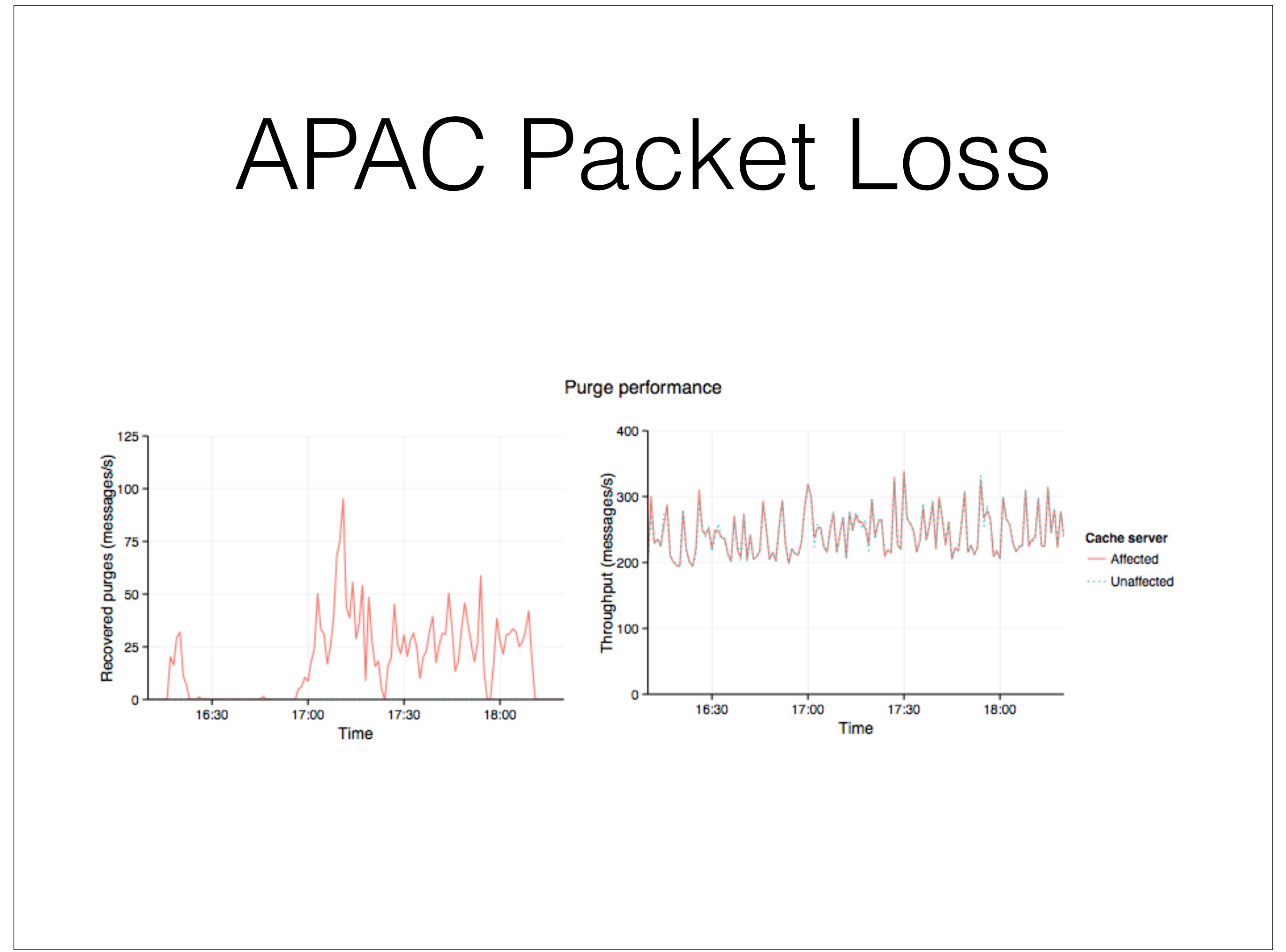


Most purges are sent from the US

Firewall Partition

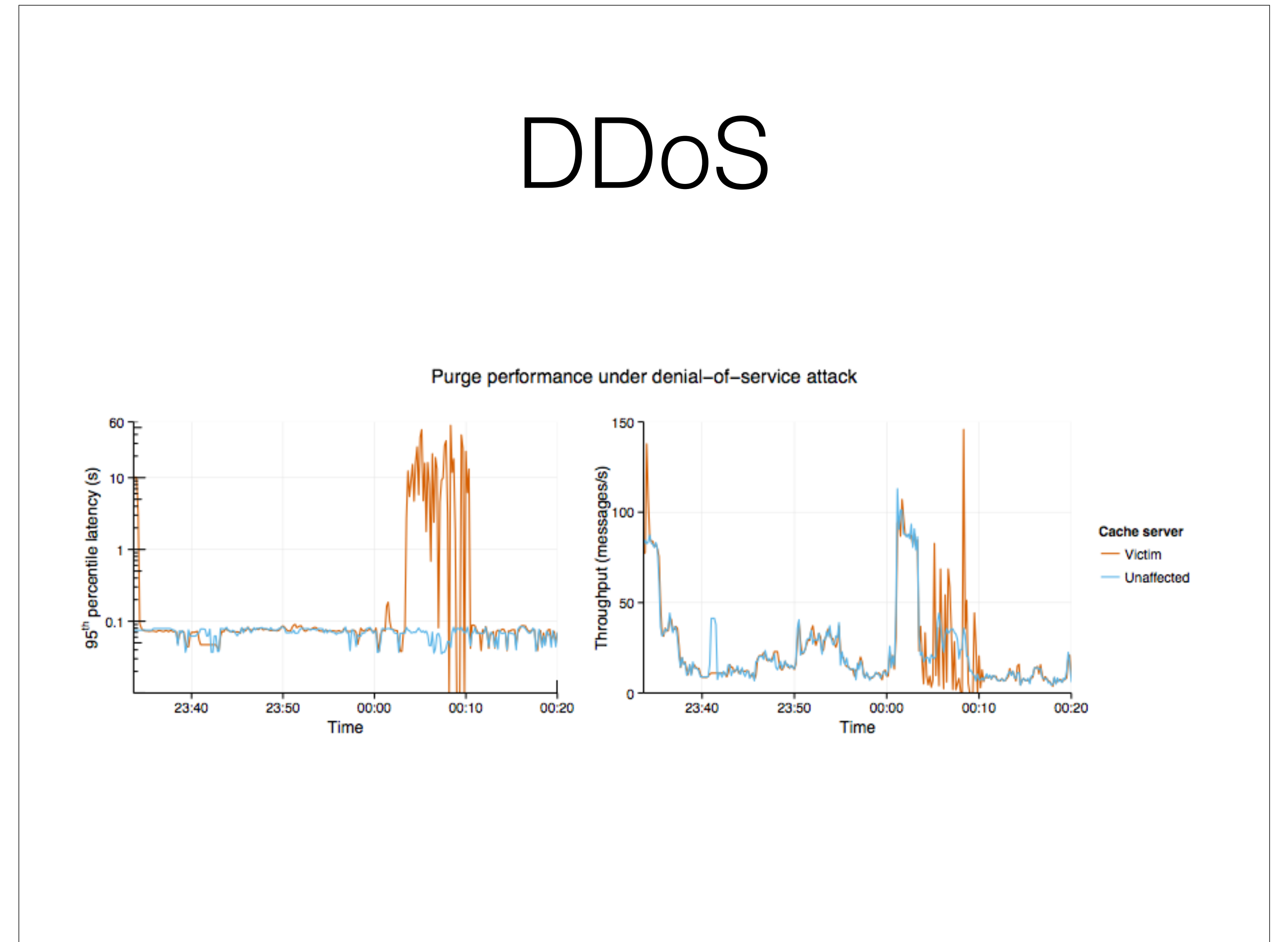


firewall misconfiguration prevented two servers (B and D) from communicating with servers outside the datacenter. A and C were unaffected.

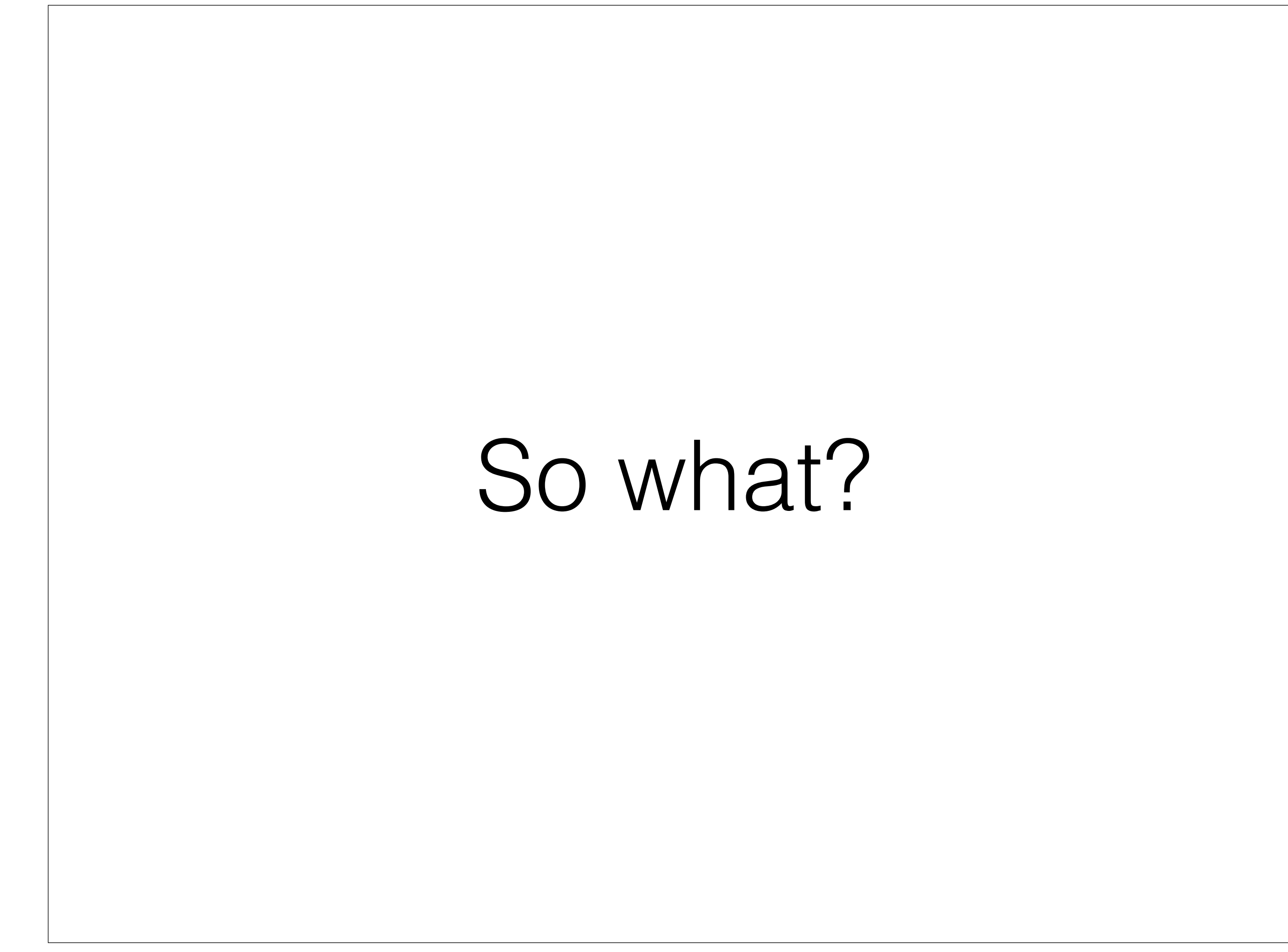


extended packet loss in APAC region for multiple hours, up to 30% at some points

no noticeable difference in throughput



The victim server was completely unreachable via ssh during the attack



CONCLUSION

- this is the system we implemented
- but why does it matter how well it works? why should you care?

BRUCE

We can go home at night, and don't need to worry about this thing failing due to network problems.

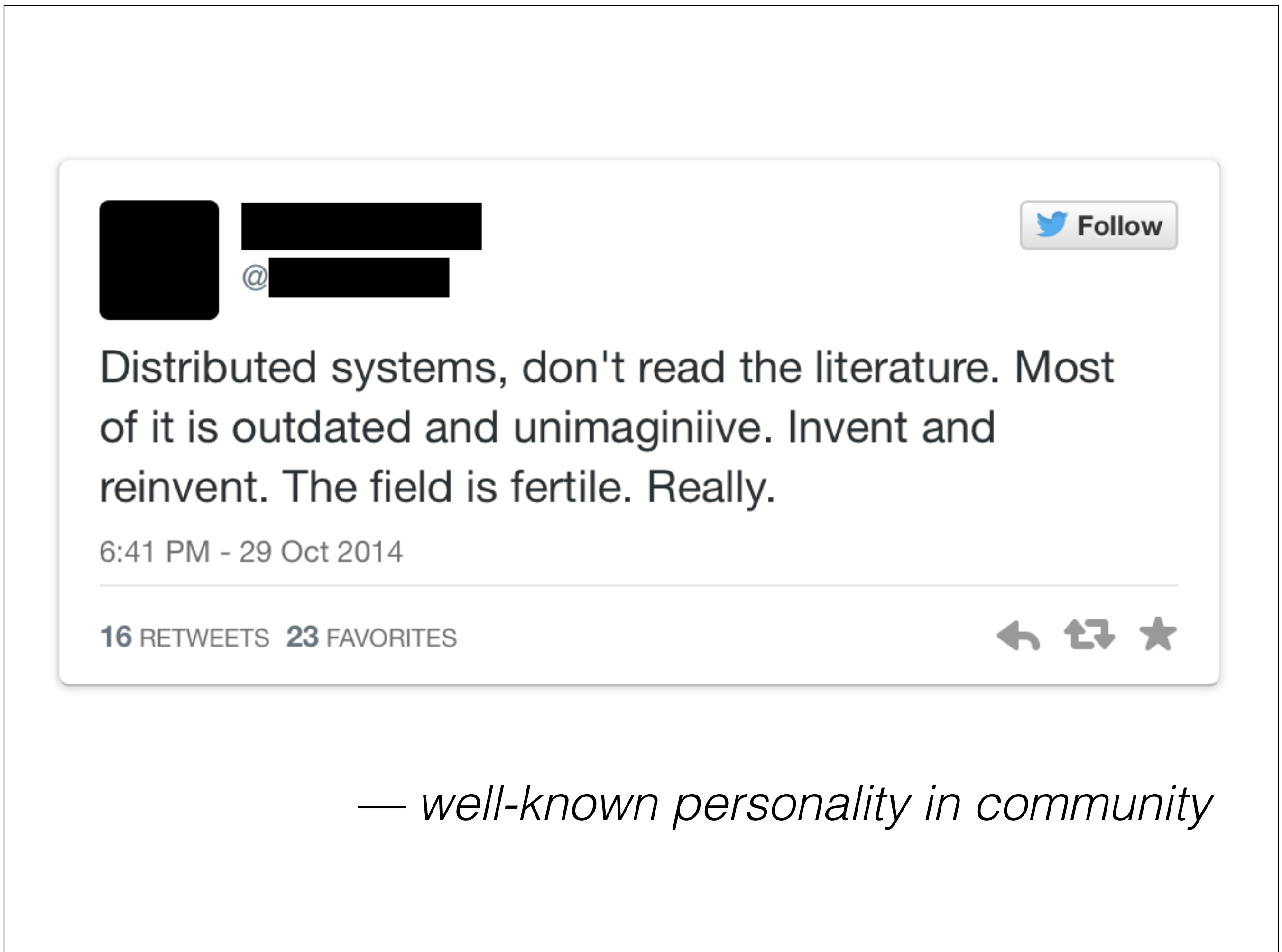
We don't have to debug distributed systems algorithms it at two in the morning.

We've been able to grow the number of purges by an order of magnitude without having to rewrite parts of the system.

etc…

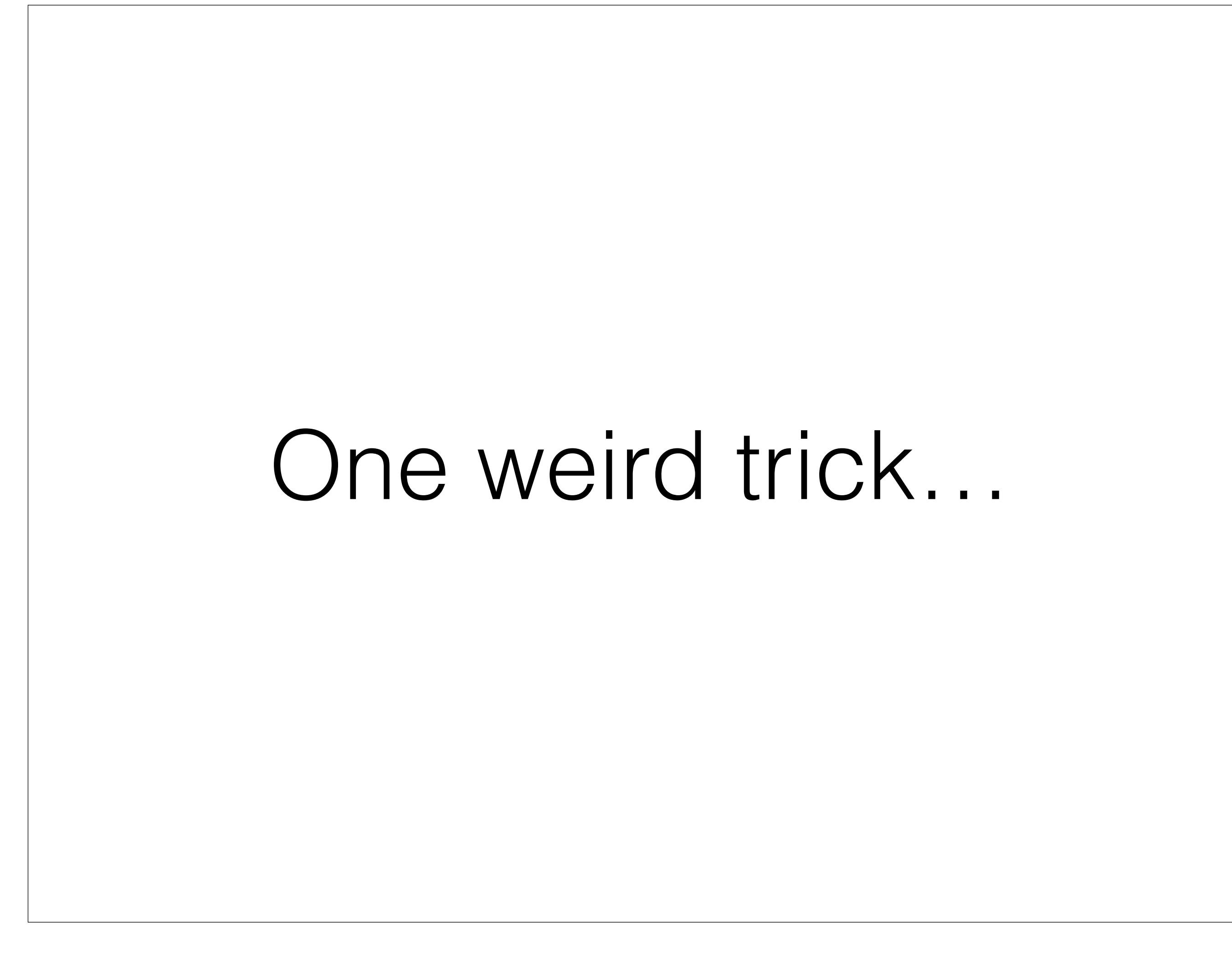
so this is great for us, but why do you care about the history of how we built our purging system?

handoff to tyler



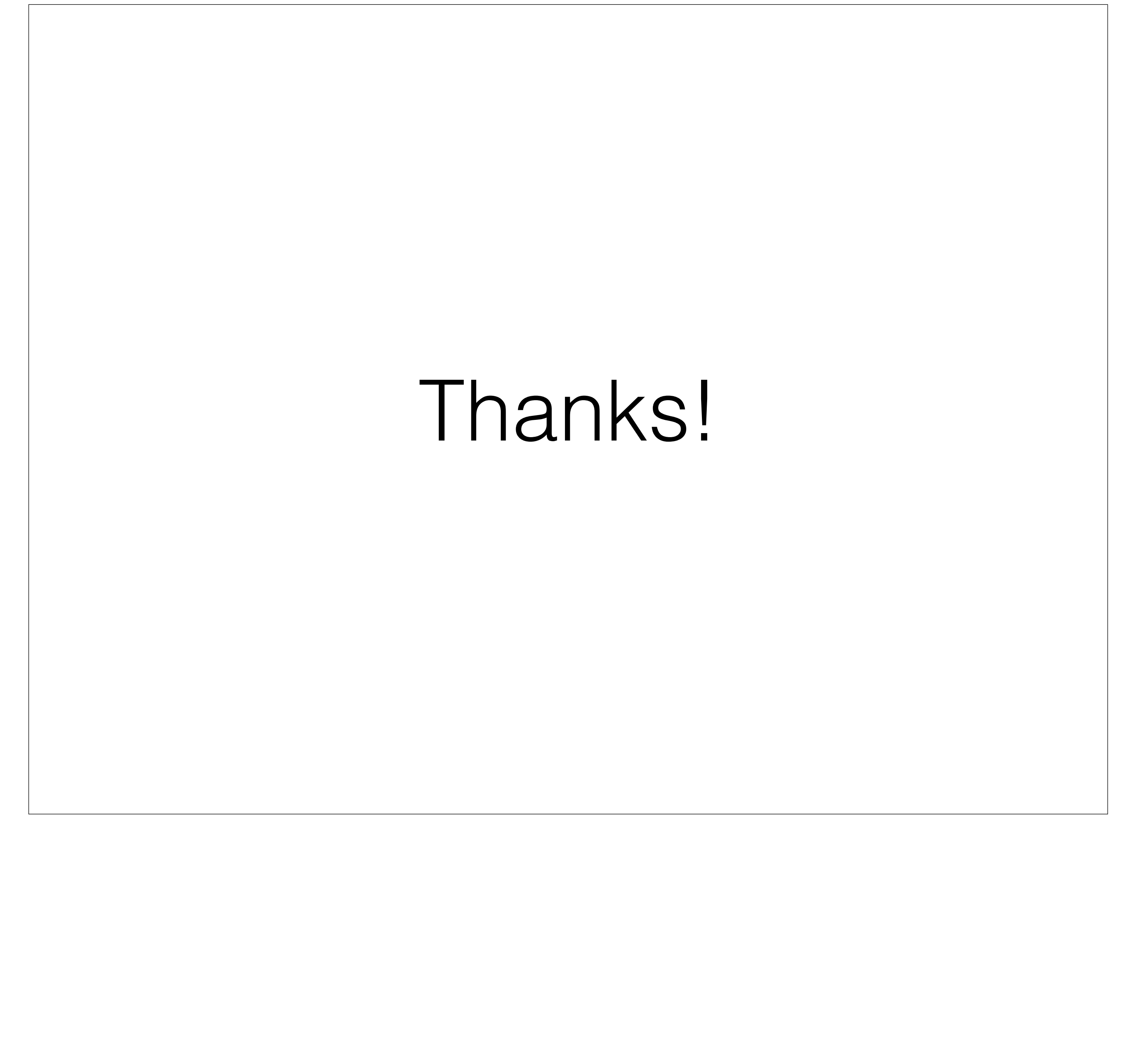
So, this was supposed to be a sponsored talk, but instead of trying to sell you on Fastly, the reason we give this talk is actually as a sort of Public Service Announcement. Don't heed advice like this. Certainly spend time inventing and thinking, but don't ignore the research.

It would have taken us quite a lot more trial and error to come to a system that we're as happy with now and long-term if we hadn't based it on solid research. And because we did, we now have a good foundation to invent new, and actually original, ideas on top of.



So, essentially, if you take away one thing from this talk, remember this one weird trick to save yourself 20 or 30 years worth of research work…

Read more papers.


Thanks!

Questions?

Come to our booth!