Challenges with distributed systems

Jacob Gabrielson

Challenges with distributed systems

The moment we added our second server, distributed systems became the way of life at Amazon. When I started at Amazon in 1999, we had so few servers that we could give some of them recognizable names like “fishy” or “online-01”. However, even in 1999, distributed computing was not easy. Then as now,

challenges with distributed systems involved latency, scaling, understanding networking APIs, marshalling and unmarshalling data, and the complexity of algorithms such as Paxos. As the systems quickly grew larger and more distributed, what had been theoretical edge cases turned into regular occurrences.

Developing distributed utility computing services, such as reliable long-distance telephone networks, or Amazon Web Services (AWS) services, is hard. Distributed computing is also *weirder* and *less intuitive* than other forms of computing because of two interrelated problems. **Independent failures** and **nondeterminism** cause the most impactful issues in distributed systems. In addition to the typical computing failures most engineers are used to, failures in distributed systems can occur in many other ways. What's worse, it's impossible always to *know* whether something failed.

Across The Amazon Builders' Library, we address how AWS handles the complicated development and operations issues arising from distributed systems. Before diving into these techniques in detail in other articles, it's worth reviewing the concepts that contribute to why distributed computing is so, well, weird. First, let's review the types of distributed systems.

Types of distributed systems

Distributed systems actually vary in difficulty of implementation. On one end of the spectrum, we have *offline* distributed systems. These include batch processing systems, big data analysis clusters, movie scene rendering farms, protein folding clusters, and the like. While far from trivial to implement, offline distributed systems get almost all of the benefits of distributed computing (scalability and fault tolerance) and almost none of the downsides (complex failure modes and non-determinism).

In the middle of the spectrum, we have *soft real-time* distributed systems. These are critical systems that must continually produce or update results, but they have a relatively generous time window in which to do so. Examples of these systems include some search index builders, systems that look for impaired servers, roles for Amazon Elastic Compute Cloud (Amazon EC2), and so forth. A search indexer might be offline for (depending on the application) from 10 minutes to many hours without undue customer impact. Roles for Amazon EC2 must push updated credentials to (essentially) every EC2 instance, but it has hours in which to do so because the old credentials don't expire for some time.

At the far, and most difficult, end of the spectrum, we have *hard real-time* distributed systems. These are often called request/reply services. At Amazon, when we think about building a distributed system, the hard real-time system is the first type that we think of. Unfortunately, hard real-time distributed systems are the most difficult to get right. What makes them difficult is that requests arrive unpredictably and responses must be given rapidly (for example, the customer is actively waiting for the response). Examples include front-end web servers, the order pipeline, credit card transactions, every AWS API, telephony, and so on. Hard real-time distributed systems are the major focus of this article.

Hard real-time systems are weird

In one plot line from the *Superman* comic books, Superman encounters an alter ego named *Bizarro* who lives on a planet (*Bizarro World*) where everything is backwards. Bizarro *looks* kind of similar to Superman, but he is actually evil. Hard real-time distributed systems are the same. They look kind of like regular computing, but are actually different, and, frankly, a bit on the evil side.

Hard real-time distributed systems development is bizarre for one reason: request/reply networking. We do *not* mean the nitty-gritty details of TCP/IP, DNS, sockets, or other such protocols. Those subjects are potentially difficult to understand, but they resemble other hard problems in computing.

What makes hard real-time distributed systems difficult is that the network enables sending messages from one *fault domain* to another. Sending a message might seem innocuous. In fact, sending messages is where everything starts getting more complicated than normal.

To take a simple example, look at the following code snippet from an implementation of Pac-Man. Intended to run on a single machine, it doesn't send any messages over any network.

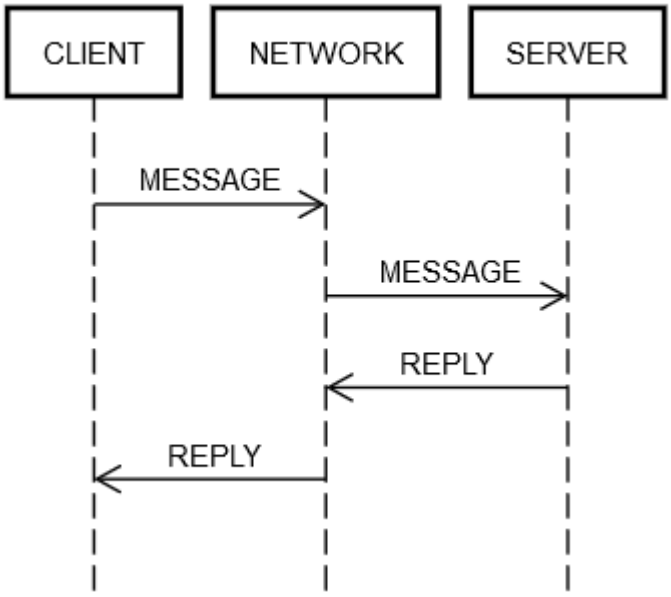
```
board.move(pacman, user.joystickDirection())
ghosts = board.findAll(":ghost")
for (ghost in ghosts)
  if board.overlaps(pacman, ghost)
    user.slayBy(":ghost")
    board.remove(pacman)
    return
```

Now, let's imagine developing a networked version of this code, where the board object's state is maintained on a separate server. Every call to the board object, such as findAll(), results in sending and receiving messages between two servers.

Whenever a request/reply message is sent between two servers, the same set of eight steps, at a minimum, must *always* happen. To understand the networked Pac-Man code, let's review the basics of request/reply messaging.

Request/reply messaging across a network

One round-trip request/reply action always involves the same steps. As shown in the following diagram, client machine CLIENT sends a request MESSAGE over network NETWORK to server machine SERVER, which replies with message REPLY, also over network NETWORK.



In the happy case where everything works, the following steps occur:

1. **POST REQUEST**: CLIENT puts request MESSAGE onto NETWORK.
2. **DELIVER REQUEST**: NETWORK delivers MESSAGE to SERVER.
3. **VALIDATE REQUEST**: SERVER validates MESSAGE.
4. **UPDATE SERVER STATE**: SERVER updates its state, if necessary, based on MESSAGE.
5. **POST REPLY**: SERVER puts reply REPLY onto NETWORK.
6. **DELIVER REPLY**: NETWORK delivers REPLY to CLIENT.

7. **VALIDATE REPLY**: CLIENT validates REPLY.
8. **UPDATE CLIENT STATE**: CLIENT updates its state, if necessary, based on REPLY.

Those are a lot of steps for one measly round trip! Still, those steps are the *definition* of request/reply communication across a network; there is no way to skip any of them. For example, it's impossible to skip step 1. The client must put MESSAGE onto NETWORK somehow. Physically, this means sending packets via a network adapter, which causes electrical signals to travel over wires through a series of routers that comprise the network between CLIENT and SERVER. This is separate from step 2 because step 2 could fail for independent reasons, such as SERVER suddenly losing power and being unable to accept the incoming packets. The same logic can be applied to the remaining steps.

Thus, a single request/reply over the network explodes *one* thing (calling a method) into *eight* things. Worse, as noted above, CLIENT, SERVER, and NETWORK can fail *independently* from each other. Engineers' code must handle *any* of the steps described earlier failing. That is rarely true of typical engineering. To see why, let's review the following expression from the single-machine version of the code.

```
board.find("pacman")
```

Technically, there are some weird ways this code could fail at runtime, even if the implementation of board.find is itself bug-free. For example, the CPU could spontaneously overheat at runtime. The machine's power supply could fail, also spontaneously. The kernel could panic. Memory could fill up, and some object that board.find attempts to create can't be created. Or, the disk on the machine it's running on could fill up, and board.find could fail to update some statistics file and then return an error, even though it probably shouldn't. A gamma ray could hit the server and flip a bit in RAM. But, most of the time, engineers don't worry about those things. For example, unit tests never cover the "what if the CPU fails" scenario, and only rarely cover out-of-memory scenarios.

In typical engineering, these types of failures occur on a single machine; that is, a single *fault domain*. For example, if the board.find method fails because the CPU spontaneously fries, it's safe to assume that the entire machine is down. It's not even conceptually possible to handle that error. Similar assumptions can be made about the other types of errors listed earlier. You could try to write tests for some of these cases, but there is little point for typical engineering. If these failures do happen, it's safe to assume that everything else will fail too. Technically, we say that they all *share fate*. Fate sharing cuts down immensely on the different failure modes that an engineer has to handle.

Handling failure modes in hard real-time distributed systems

Engineers working on hard real-time distributed systems must test for all aspects of network failure because the servers and the network do *not* share fate. Unlike the single machine case, if the network fails, the client machine will keep working. If the remote machine fails, the client machine will keep working, and so forth.

To exhaustively test the failure cases of the request/reply steps described earlier, engineers must assume that each step could fail. And, they must ensure that code (on both client and server) always behaves correctly in light of those failures.

Let's look at a round-trip request/reply action where things aren't working:

1. **POST REQUEST fails**: Either network NETWORK failed to deliver the message (for example, intermediate router crashed at just the wrong moment), or SERVER rejected it explicitly.

2. **DELIVER REQUEST fails**: NETWORK successfully delivers MESSAGE to SERVER, but SERVER crashes right after it receives MESSAGE.
3. **VALIDATE REQUEST fails**: SERVER decides that MESSAGE is invalid. The cause can be almost anything. For example, corrupted packets, incompatible software versions, or bugs on either client or server.
4. **UPDATE SERVER STATE fails**: SERVER tries to update its state, but it doesn't work.
5. **POST REPLY fails**: Regardless of whether it was trying to reply with success or failure, SERVER could fail to post the reply. For example, its network card might fry just at the wrong moment.
6. **DELIVER REPLY fails**: NETWORK could fail to deliver REPLY to CLIENT as outlined earlier, even though NETWORK was working in an earlier step.
7. **VALIDATE REPLY fails**: CLIENT decides that REPLY is invalid.
8. **UPDATE CLIENT STATE fails**: CLIENT could receive message REPLY but fail to update its own state, fail to understand the message (due to being incompatible), or fail for some other reason.

These failure modes are what make distributed computing so hard. I call them the *eight failure modes of the apocalypse*. In light of these failure modes, let's review this expression from the Pac-Man code again.

```
board.find("pacman")
```

This expression expands into the following client-side activities:

1. Post a message, such as {action: "find", name: "pacman", userId: "8765309"}, onto the network, addressed to the Board machine.
2. If the network is unavailable, or the connection to the Board machine is explicitly refused, raise an error. This case is somewhat special because the client knows, deterministically, that the request could not possibly have been received by the server machine.
3. Wait for a reply.
4. If a reply is never received, time out. In this step, timing out means that the result of the request is UNKNOWN. It may or may not have happened. The client must handle UNKNOWN correctly.
5. If a reply is received, determine if it's a success reply, error reply, or incomprehensible/corrupt reply.
6. If it isn't an error, unmarshall the response and turn it into an object the code can understand.
7. If it is an error or incomprehensible reply, raise an exception.
8. Whatever handles the exception has to determine if it should *retry* the request or give up and stop the game.

The expression also starts the following server-side activities:

1. Receive the request (this may not happen at all).
2. Validate the request.
3. Look up the user to see if the user is still alive. (The server might have given up on the user because it hadn't received any messages from them for too long.)
4. Update the keep-alive table for the user so the server knows they're (probably) still there.
5. Look up the user's position.
6. Post a response containing something like {xPos: 23, yPos: 92, clock: 23481984134}.
7. Any further server logic must correctly handle the future effects of the client. For example, failing to receive the message, receiving it but not understanding it, receiving it and crashing, or handling it successfully.

In summary, *one* expression in normal code turns into *fifteen* extra steps in hard real-time distributed systems code. This expansion is due to the eight different points at which each round-trip communication between client and server can fail. Any expression that represents a round trip over the network, such as board.find("pacman"), results in the following.

```
(error, reply) = network.send(remote, actionData)
switch error
  case POST_FAILED:
    // handle case where you know server didn't get it
  case RETRYABLE:
    // handle case where server got it but reported transient failure
  case FATAL:
    // handle case where server got it and definitely doesn't like it
  case UNKNOWN: // i.e., time out
    // handle case where the *only* thing you know is that the server received
    // the message; it may have been trying to report SUCCESS, FATAL, or RETRYABLE
  case SUCCESS:
    if validate(reply)
      // do something with reply object
    else
      // handle case where reply is corrupt/incompatible
```

This complexity is unavoidable. If code doesn't handle all cases correctly, the service will eventually fail in bizarre ways. Imagine trying to write tests for all the failure modes a client/server system such as the Pac-Man example could run into!

Testing hard real-time distributed systems

Testing the single-machine version of the Pac-Man code snippet is comparatively straightforward. Create some different Board objects, put them into different states, create some User objects in different states, and so forth. Engineers would think hardest about edge conditions, and maybe use generative testing, or a fuzzer.

In the Pac-Man code, there are four places where the board object is used. In distributed Pac-Man, there are four points in that code that have five different possible outcomes, as illustrated earlier (POST_FAILED, RETRYABLE, FATAL, UNKNOWN, or SUCCESS). These multiply the state space of tests tremendously. For example, engineers of hard real-time distributed systems have to handle many permutations. Say that the call to board.find() fails with POST_FAILED. Then, you have to test what happens when it fails with RETRYABLE, then you have to test what happens if it fails with FATAL, and so on.

But even that testing is insufficient. In typical code, engineers may assume that if board.find() works, then the next call to board, board.move(), will also work. In hard real-time distributed systems engineering, there is no such guarantee. The server machine could fail independently at any time. As a result, engineers have to write tests for all five cases for *every* call to board. Let's say an engineer came up with 10 scenarios to test in the single-machine version of Pac-Man. But, in the distributed systems version, they have to test each of those scenarios *20* times. Meaning that the test matrix balloons from 10 to 200!

But, wait, there's more. The engineer may *also* own the *server* code as well. Whatever combination of client, network, and server side errors occur, they must test so that the client and the server don't end up

in a corrupted state. The server code might look like the following.

```
handleFind(channel, message)
 if !validate(message)
  channel.send(INVALID_MESSAGE)
  return
 if !userThrottle.ok(message.user())
  channel.send(RETRYABLE_ERROR)
  return
 location = database.lookup(message.user())
 if location.error()
  channel.send(USER_NOT_FOUND)
  return
 else
  channel.send(SUCCESS, location)

handleMove(...)
 ...

handleFindAll(...)
 ...

handleRemove(...)
 ...
```

There are four server-side functions to test. Let's assume that each function, on a single machine, has five tests each. That's 20 tests right there. Because clients send multiple messages to the same server, tests should simulate sequences of different requests to make sure that the server remains robust. Examples of requests include find, move, remove, and findAll.

Let's say one construct has 10 different scenarios with an average of three calls in each scenario. That's 30 more tests. But, one scenario also needs to test failure cases. For each of those tests, you need to simulate what happens if the client received any of the four failure types (POST_FAILED, RETRYABLE, FATAL, and UNKNOWN) and then calls the server again with an invalid request. For example, a client might successfully call find, but then sometimes get UNKNOWN back when it calls move. It might then call find again for some reason. Does the server handle this case correctly? Probably, but you won't know unless you test for it. So, as with the client-side code, the test matrix on the server side explodes in complexity as well.

Handling unknown unknowns

It is mind-boggling to consider all the permutations of failures that a distributed system can encounter, especially over multiple requests. One way we've found to approach distributed engineering is to distrust *everything*. Every line of code, unless it could not possibly cause network communication, might not do what it's supposed to.

Perhaps the hardest thing to handle is the UNKNOWN error type outlined in the earlier section. The client doesn't always know if the request succeeded. Maybe it *did* move Pac-Man (or, in a banking service, withdraw money from the user's bank account), or maybe it *didn't*. How should engineers handle such things? It's difficult because engineers are human, and humans tend to struggle with true uncertainty.

Humans are used to looking at code like the following.

```
bool isEven(number)
  switch number % 2
    case 0
      return true
    case 1
      return false
```

Humans understand this code because it does what it *looks like* it does. Humans struggle with the distributed version of the code, which distributes some of the work to a service.

```
bool distributedIsEven(number)
  switch mathServer.mod(number, 2)
    case 0
      return true
    case 1
      return false
    case UNKNOWN
      return WHAT_THE_FARG?
```

It's almost impossible for a human to figure out how to handle UNKNOWN correctly. What does UNKNOWN really mean? Should the code retry? If so, how many times? How long should it wait between retries? It gets even worse when code has side-effects. Inside of a budgeting application running on a single machine, withdrawing money from an account is easy, as shown in the following example.

```
class Teller
  bool doWithdraw(account, amount)
    switch account.withdraw(amount)
      case SUCCESS
        return true
      case INSUFFICIENT_FUNDS
        return false
```

However, the distributed version of that application is weird because of UNKNOWN.

```
class DistributedTeller
  bool doWithdraw(account, amount)
    switch this.accountService.withdraw(account, amount)
      case SUCCESS
        return true
      case INSUFFICIENT_FUNDS
        return false
      case UNKNOWN
        return WHAT_THE_FARG?
```

Figuring out how to handle the UNKNOWN error type is one reason why, in distributed engineering, *things are not always as they seem.*

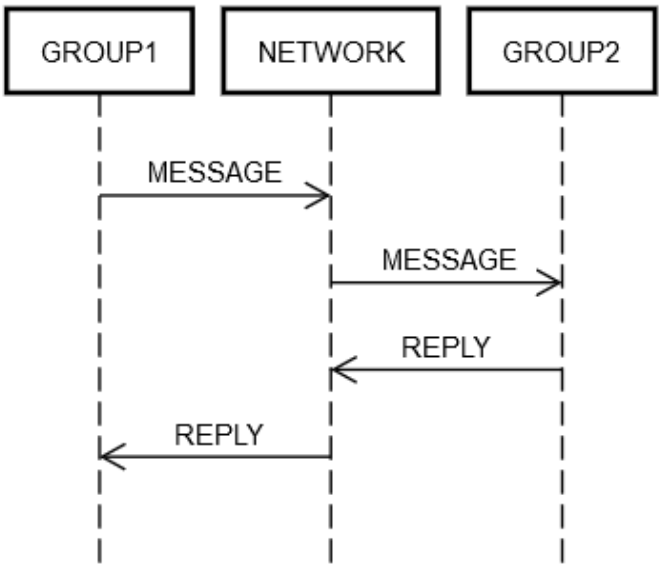
Herds of hard real-time distributed systems

The eight failure modes of the apocalypse can happen at any level of abstraction within a distributed system. The earlier example was limited to a single client machine, a network, and a single server machine. Even in that simplistic scenario, the failure state matrix exploded in complexity. Real distributed systems have more complicated failure state matrices than the single client machine example. Real distributed systems consist of multiple machines that may be viewed at multiple levels of abstraction:

1. Individual machines
2. Groups of machines
3. Groups of groups of machines
4. And so on (potentially)

For example, a service built on AWS might group together machines dedicated to handling resources that are within a particular Availability Zone. There might also be two more groups of machines that handle two other Availability Zones. Then, those groups might be grouped into an AWS Region group. And that Region group might communicate (logically) with other Region groups. Unfortunately, even at this higher, more logical level, *all the same problems apply*.

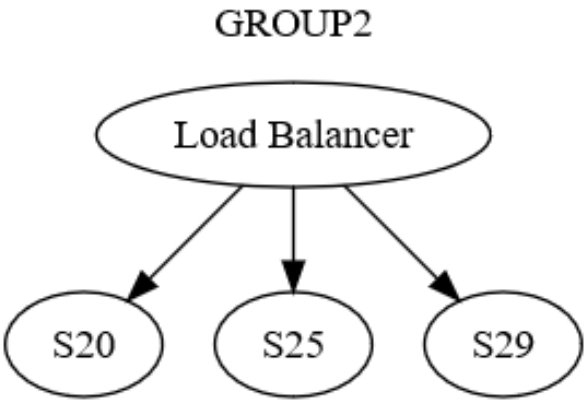
Let's assume a service has grouped some servers into a single logical group, GROUP1. Group GROUP1 might sometimes send messages to another group of servers, GROUP2. This is an example of *recursive distributed engineering*. All the same networking failure modes described earlier can apply here. Say that GROUP1 wants to send a request to GROUP2. As shown in the following diagram, the two-machine request/reply interaction is just like that of the single machine discussed earlier.



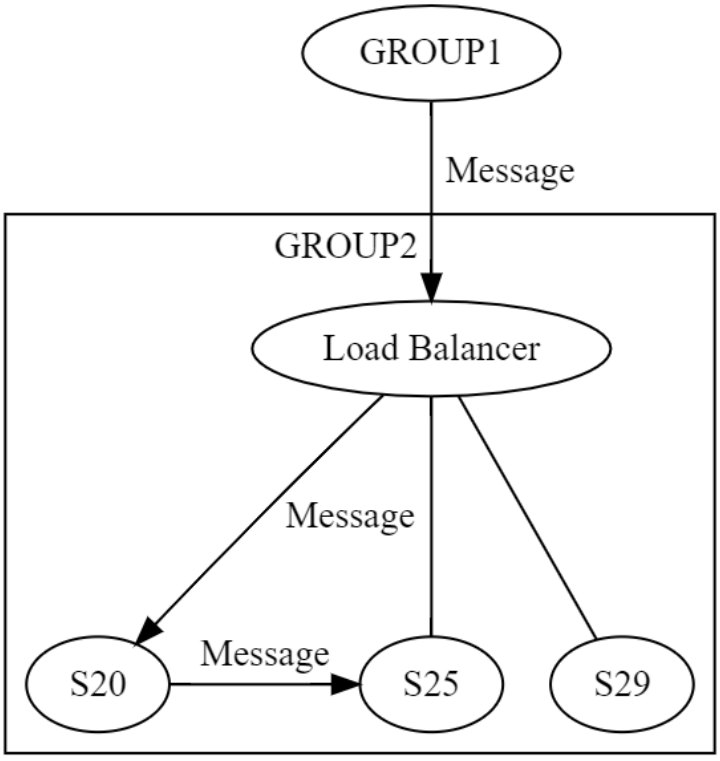
One way or another, some machine within GROUP1 has to put a message on the network, NETWORK, addressed (logically) to GROUP2. Some machine within GROUP2 has to process the request, and so forth. The fact that GROUP1 and GROUP2 are comprised of groups of machines doesn't change the fundamentals. GROUP1, GROUP2, and NETWORK can still fail independently of each other.

However, that is just the group-level view. There is also machine-to-machine level interaction within each

group. For example, GROUP2 might be structured as shown in the following diagram.



At first, a message to GROUP2 is sent, via the load balancer, to one machine (possibly S20) within the group. The designers of the system know that S20 might fail during the UPDATE STATE phase. As a result, S20 may need to pass the message to at least one other machine, either one of its peers or a machine in a different group. How does S20 actually do this? By sending a request/reply message to, say, S25, as shown in the following diagram.



Thus, S20 is performing networking recursively. All the same eight failures can occur, independently, again. Distributed engineering is happening twice, instead of once. The GROUP1 to GROUP2 message, at the logical level, can fail in all eight ways. That message results in another message, which can itself fail, independently, in all the eight ways discussed earlier. Testing this scenario would involve, at least the following:

- A test for all eight ways GROUP1 to GROUP2 group-level messaging can fail.
- A test for all eight ways S20 to S25 server-level messaging can fail.

This request/reply messaging example shows why testing distributed systems remains an especially vexing problem, even after over 20 years of experience with them. Testing is challenging given the vastness of edge cases, but it's especially important in these systems. Bugs can take a long time to surface after systems are deployed. And, bugs can have an unpredictably broad impact to a system and its adjacent systems.

Distributed bugs are often latent

If a failure is going to happen eventually, common wisdom is that it's better if it happens sooner rather than later. For example, it's better to find out about a scaling problem in a service, which will require six months to fix, *at least* six months before that service will have to achieve such scale. Likewise, it's better to find bugs before they hit production. If the bugs do hit production, it's better to find them quickly, before they affect many customers or have other adverse effects.

Distributed bugs, meaning, those resulting from failing to handle all the permutations of eight failure modes of the apocalypse, are often severe. Examples over time abound in large distributed systems, from telecommunications systems to core internet systems. Not only are these outages widespread and expensive, they can be caused by bugs that were deployed to production months earlier. It then takes a while to trigger the combination of scenarios that actually lead to these bugs happening (and spreading across the entire system).

Distributed bugs spread epidemically

Let me describe another problem that is fundamental to distributed bugs:

1. Distributed bugs necessarily involve use of the network.
2. Therefore, distributed bugs are more likely to spread to other machines (or groups of machines), because, by definition, they *already involve the only thing that links machines together*.

Amazon has experienced these distributed bugs, too. An old, but relevant, example is a site-wide failure of www.amazon.com. The failure was caused by a single server failing within the remote catalog service when its disk filled up.

Due to mishandling of that error condition, the remote catalog server started returning empty responses to every request it received. It also started returning them very quickly, because it's a lot faster to return nothing than something (at least it was in this case). Meanwhile, the load balancer between the website and the remote catalog service didn't notice that all the responses were zero-length. But, it *did* notice that they were blazingly faster than all the other remote catalog servers. So, it sent a huge amount of the traffic from www.amazon.com to the one remote catalog server whose disk was full. Effectively, the entire website went down because one remote server couldn't display any product information.

We found the bad server quickly and removed it from service to restore the website. Then, we followed up with our usual process of determining root causes and identifying issues to prevent the situation from happening again. We shared those lessons across Amazon to help prevent other systems from having the same problem. In addition to learning the specific lessons about this failure mode, this incident served as a great example of how failure modes propagate quickly and unpredictably in distributed systems.

Summary of problems in distributed systems

In short, engineering for distributed systems is hard because:

- Engineers can't combine error conditions. Instead, they must consider many permutations of failures. Most errors can happen at any time, independently of (and therefore, potentially, in combination with) any other error condition.
- The result of any network operation can be UNKNOWN, in which case the request may have succeeded, failed, or received but not processed.
- Distributed problems occur at *all* logical levels of a distributed system, not just low-level physical machines.
- Distributed problems get worse at higher levels of the system, due to recursion.
- Distributed bugs often show up long after they are deployed to a system.
- Distributed bugs can spread across an entire system.
- Many of the above problems derive from the laws of physics of networking, which can't be changed.

Just because distributed computing is hard—and weird—doesn't mean that there aren't ways to tackle these problems. Throughout the Amazon Builders' Library, we dig into how AWS manages distributed systems. We hope you'll find some of what we've learned valuable as you build for your customers