Byzantine Fault Isolation in the Farsite Distributed File System

Howdy. My name's John Douceur. I'm with Microsoft Research, and this talk is about some work of Jon Howell's and mine, developing the technique of Byzantine fault isolation in the context of the Farsite distributed file system.

Definitions

As you may well know...

...a Byzantine fault is a failure of a system component that produces arbitrary behavior. Unlike stopping faults and step-omission faults that fail in a more predictable fashion, a Byzantine fault can do pretty much anything; it's a good model for viruses, hackers, and malicious users with physical access to their machines.

Byzantine fault isolation, the focus of this talk, is a methodology we've developed for designing a distributed system that can, under Byzantine failure, operate with application-defined partial correctness.

And I'll often abbreviate that as BFI.

We developed BFI in the context of Farsite, a serverless distributed file system designed to be scalable, strongly consistent, and secure despite the fact that it runs on an untrusted infrastructure that is highly susceptible to Byzantine faults.

Talk Outline

So I'm going to begin by describing the Farsite system, to give you some context. I'll describe the technique of Byzantine fault tolerance and explain why it doesn't scale. I'll describe how Farsite scales by using multiple BFT groups, and how that gives rise to the need for isolating Byzantine faults. I'll talk about formal system specification, which is the tool we use for designing a system that can isolate Byzantine faults. And lastly, I'll describe Byzantine fault isolation in Farsite.

Farsite System

Farsite is software system that takes a bunch of PCs sitting on people's desks…

…and pulls them into a logically centralized file server.

It's just a virtual file server; there's no physically centralized anything, but it can be used just like a real physical file server.

A client machine, acting on behalf of its local user…

…can place a file into the virtual file server.

The system encrypts the file, for security...

...makes multiple replicas of the file contents, for availability…

…and distributes those replicas among several machines, which act as servers for this file. There's nothing special about these machines; they also act as clients for their local users; here, they're acting in their server capacity.

When another machine, in its capacity as a client…

…tries to open the file…

…it can get a copy of the contents from another machine.

If the user on the client has read access to the file, he can decrypt it to access the file contents. That's all I'm going to say about file content. What's relevant to this talk...

Farsite System – Metadata

...is the way we manage metadata. Unlike file content, which is opaque, metadata has to be understandable by the system, so we can't rely on simple encryption for security.

Instead, Farsite assembles a small set of machines into a Byzantine-fault-tolerant replicated state machine, or BFT group.

The file-system metadata...

... is replicated onto all of the members of the BFT group.

And for illustration, we'll assume these other machines act as clients…

…meaning that they act on behalf of their local users.

The members of the BFT group have local users as well; they're just normal PCs sitting on people's desks. A user isn't supposed to get involved with the server-side behavior of his machine.

However, some users might be malicious. And a malicious user can do pretty much anything he wants to his own machine.

So, when a user wants to perform an operation…

…her client replicates the request to all the machines in the BFT group.

The group uses a Byzantine agreement protocol to assign sequence numbers to messages; this establishes a processing order.

The bad guy might try to disrupt the agreement by sending different values or not sending any value.

But a result going back twenty-five years says that you can still reach agreement in the presence of T faults among R replicas…

…if and only if R is strictly greater than 3 T. So with four machines, we can tolerate that one malicious guy.

Next, the group uses a two-step prepare-commit protocol, such that each member will only commit once it knows that at least 2 T other members are ready to commit.

Again, the bad guy might try to disrupt the protocol, but with enough good guys present, they'll all commit.

The members then deterministically update their metadata, meaning that the result of the update is solely a function of the previous state and the request received from the client. This combination of consistent ordering of requests, two-step supermajority commit, and deterministic state update ensures that all correct members reach the same result.

The group sends replies to the client.

And again, the bad guy might send a different reply or not reply at all, but the client can be sure when it gets enough matching replies that the result is correct.

The Cost of BFT Groups

BFT groups like that are wonderful, but they come at a substantial cost...

... compared to running a service on a single machine.

In terms of computation...

...rather than computing everything once...

...you have to compute everything four times, or, for an N-member group, N times.

There's also a cost in terms of message delays...

...and total message traffic.

When a client talks to a single, non-replicated server...

...it sends a request...

...and it gets a reply.

That's two message delays...

...and two messages. When talking to a four-member BFT group...

...there's the request, then...

...one...

...two...

...three phases of communication among the servers...

...and the reply.

That's a total of five message delays, and if you were counting the little arrows flying around...

...32 messages. There are techniques to bring these numbers down somewhat. But all known mechanisms that tolerate Byzantine faults...

Throughput vs. Scale

...have poor throughput versus scale. When distributed-systems folks call a system "scalable"...

...we ideally mean that system throughput increases linearly with scale. N machines give you N times the throughput.

Typically, we're accustomed to rather less than linear speedup, for various reasons.

But we don't commonly consider a system "scalable" if adding machines fails to improve its throughput.

BFT groups are even worse. Adding machines actually decreases throughput, because the purpose of those machines is for increased fault tolerance, not increased performance.

Workload Sharing

But in a system like Farsite...

...every new machine that joins the system...

...in its capacity as a client...

...adds to the total workload of the system. And therefore...

...in its capacity as a server...

...it has to relieve the system of a comparable share of the total workload.

BFT at Scale

That's why we can't just take all the machines in the system...

...and pull them into one big BFT group.

The total workload of the system would get replicated onto...

...every machine, so there'd be no performance improvement...

...and there'd be a massive amount of communication among all the machines, so we'd actually see a performance degradation.

Multiple BFT Groups

So instead, we organize the machines into a bunch of small, separate BFT groups, where the group size is constant with respect to system scale.

The system workload...

...is partitioned...

...and divided among the BFT groups...

...each of which replicates its portion for fault tolerance.

Tree of BFT Groups

In Farsite, these groups are organized into a tree. That was not an arbitrary decision.

It reflects the fact that the file system itself is a tree of directories and files.

The namespace tree is partitioned into regions, each of which is managed by a BFT group, such that...

...the tree of groups overlays the dominant structure of the namespace tree.

Delegation to New Group

I made it sound like that partitioning is global and static, but it actually evolves organically.

If a particular BFT group gets overloaded, it can grab some other machines in the system...

... assemble them into a new BFT group...

...and delegate a portion of its metadata to this new group.

Pathname Resolution

As an example of how these groups collaborate....

...consider a client...

...that wants to resolve a pathname. In the most basic form of the protocol...

...the client asks the root group to resolve the path.

...the root group responds with as much information as it knows, telling the client that "users Alice" is managed by this other BFT group.

The client contacts this next group...

...which directs the client to yet another group.

And when the client contacts this group...

...it resolves the path. A consequence of this architecture is that...

...a Byzantine-faulty group anywhere in the tree...

...can lie about the metadata it manages and thereby corrupt path-based operations...

...to files further down in the namespace, which, as far is the client can tell...

...is a lot like failures in the descendent BFT groups. Now, why am I even talking about group failure?

Machine Failures at Scale

Isn't the whole point of BFT groups to hide machine failure? Well, sure, but the more machines you have...

...the more failed machines you're likely to have, for a given rate of machine failure. And when we pull these machines together into BFT groups...

Group Failures at Scale

...there's an increasing probability with system scale that somewhere in the system...

...there'll be at least one group that has more machine failures than it can handle.

System Failure at Scale

And because these groups work together as a system, a single faulty group...

...can spread its corruption to other groups, and thence to other groups, eventually corrupting the entire system, unless there is some mechanism in place to quarantine Byzantine faults.

Quantitative Fault Analysis

Let's look at that quantitatively. For an example system, I'll use Farsite. To keep the analysis simple, we'll assume that files are partitioned evenly among the BFT groups, and we'll assume that machine failures are independent. For concreteness, I'll take the probability that a machine is Byzantine-faulty as one in a thousand; the paper talks about what happens when you vary this value. We're going to evaluate the operational fault rate, which is: If you select a file uniformly at random and perform an operation on that file, what's the probability that the operation will exhibit a fault.

Operational Faults vs. System Scale

With a machine fault probability of 10^{-3}...

...a single, 4-member BFT group has an operational fault rate of 6×10^{-6}. That's pretty good; it's better than 5 nines.

But with no fault isolation, as the system scale increases, that fault probability rises.

When we get to a scale of a hundred thousand, nearly half of all operations exhibit faults, in expectation. That's unusable.

If fault isolation were perfect, the fault rate would be independent of system scale...

...still 6×10^{-6}.

In Farsite, faults can spread down the BFT tree...

...so the fault rate is a function of the tree fanout. With a fanout of 16...

...the fault rate is 3×10^{-5}, which is quite good. You might say, "Yeah, but who needs BFI? Why not just increase the group size?" Well, you can.

BFT groups of size 7 will be better...

...and groups of size 10 will be competitive.

BFI versus no BFI

However, comparing...

...two different ways of building a system: 4-member BFT groups with BFI versus 10-member groups without...

...the difference in computational redundancy is 4 versus 10...

...and the difference in message traffic is 32 versus 200. So, depending on which resource is the bottleneck, increasing the group size from 4 to 10...

...reduces your system throughput by anywhere from...

...sixty percent...

...to eighty-four percent. That's dramatic.

BFI via Formal Specification

Okay, I've just spent a lot of time arguing the benefits of isolating Byzantine faults. Here's an outline of how we go about it by making use of formal system specification.

The first step is to write a semantic specification, which formally describes the way the system looks to its users.

Then, you write a distributed-system spec, which describes the way the system looks to its designers.

Each of these specs describes the system state...

...and the actions that modify that state.

Next, you construct something called a refinement, which is a formal correspondence between the two specs. In particular, the refinement describes how to interpret the distributed-system spec in semantic terms. To use this formal framework for BFI...

...you modify the distributed-system spec to expresses the behavior of Byzantine-faulty components.

That modification will cause the refinement to break. You fix it by doing two things.

Modifying the semantic spec to express how Byzantine faults manifest at the semantic level, and...

...improving the design of the distributed system...

...so as to minimize the semantic impact of distributed-system faults. I will now elaborate on all that.

Farsite Semantic Spec

Farsite is a file system, so its semantic spec is just a formal description of a file system. The semantic state includes...

...directories...

...files (and we mostly ignore the distinction between the two)...

...the parent-child links that tie the namespace together...

...a set of open handles, and what they refer to...

...and a set of pending operations. That's the state. As for semantic actions, there are two general classes: starting a new operation...

...which adds it to the pending set, and completing an operation. For instance, that open operation you see in the pending set, that operation completes by...

...resolving a path, and if the path resolves...

...creating a new handle...

...binding that handle to the indicated file...

...and removing the operation from the pending set. Very intuitive.

Farsite Distributed-System Spec

In the distributed-system spec, the state includes...

...machines...

...BFT groups of machines, which we treat monolithically...

...abstract data structures maintained by machines and groups...

...and the set of messages currently in flight...

...where the sources and destinations are established.

That's the state. A typical distributed-system action involves...

...receiving a message...

...updating local state...

...and sending another message.

Farsite Refinement

We refine the distributed-system state to semantic state by, for example, saying...

...a particular data structure on a particular group corresponds to a semantic file...

...and some other data structure on some machine corresponds to an open handle.

Actions in the distributed system might or might not have a semantic interpretation, depending on whether they're foreground or background actions.

Modifying a particular data structure might correspond to...

...starting a new operation; that's a foreground action.

Sending a message is a background action. It corresponds to a non-action in the semantic spec, meaning that there's no change at all to the semantic state.

Actions are State Transitions

The legal behavior of the distributed system is described inductively.

There's a defined initial state...

...and a set of all possible actions...

...each of which takes the system from one legal state to another. As you can see in this animation, any particular sequence of actions gives rise to a particular behavior of the system. But it's all possible sequences of these actions that define the distributed system.

The semantic spec also has a defined initial state...

...and a much simpler set of actions...

...which take it from one legal state to the next.

Proving Refinement Inductively

To prove that the distributed-system spec refines to the semantic spec, we use induction.

First, we show that the initial state of the distributed system refines to the semantic initial state. That's the basis step.

For the inductive step, we show that every foreground action in the distributed-system spec refines to an action in the semantic spec...

...and every background action in the distributed-system spec refines to a non-action in the semantic spec.

As you can see in this animation, the majority of distributed-system actions tend to be background actions. In fact, in Farsite, every action in the network and every action on a BFT group is a background action. As a point of design, foreground actions occur only on clients.

Refinement with Byzantine Faults

Now we'll modify the distributed-system spec to allow Byzantine-faulty groups.

When a group becomes faulty...

...that group's local state becomes irrelevant...

...and it's free to send any messages it wants to other machines and other BFT groups. These distributed-system actions don't refine to any action or non-action in the semantic spec. So what do we do? First, we modify the semantic spec by...

...flagging certain files as "tainted."

In the refinement, we assert that a file is tainted if and only if the group managing that file is Byzantine-faulty.

Semantic Fault Specification

And we modify the semantic spec to treat tainted files specially. In particular...

...a tainted file may have arbitrary contents and attributes. Nothing we can do about that.

–

It may appear not linked into the namespace.

And in the case of a directory, its descendents become unlinked as well.

–

It may pretend not to have children it actually has.

–

It may pretend to have children that do not exist.

–

It may pretend another tainted file is its child or parent.

–

And any operation involving a tainted file may not complete.

That's an exhaustive list of the semantics of what can go wrong. In particular, note that...

...a tainted file cannot falsely claim an existing non-tainted file as its child...

...or parent, so a fault can't break out of the subtree it's in. The way we prevent that from happening...

Distributed-System Improvements

...is by making several improvements to the distributed system.

A key improvement was to maintain some redundant information across BFT group boundaries. For example...

...a group knows the children of every file it manages...

...but a group also knows the parent of every file it manages. This prevents groups from promulgating certain lies that could violate our fault semantics.

We augment messages with information that justifies their correctness.

So a message doesn't just say, "Here's some information."

It also says, "Here's why you should believe this."

This is somewhat akin to proof-carrying code.

We ensure unambiguous chains of authority over data.

In our early designs...

...when a group delegated authority to another group...

...which in turn delegated to a third group...

... we had some mechanisms that could convey that authority while bypassing that middle group. It turned out that ambiguity in these two different channels led to Byzantine vulnerabilities...

...so we now have only one way authority is conveyed.

For operations involving multiple BFT groups, we have to carefully order our messages and state updates, to prevent a faulty group from causing a semantically visible disagreement among non-faulty groups.

One group updates its state...

...then tells two other groups, which perform different roles in the protocol, about its update.

One of those groups updates its state first...

...and then tells the other group, which waits until it hears both messages...

...before updating its own state. Which group waits for which is bound up with the semantics of the operation, and a particularly involved example of this is described in the paper.

These are some specifics we had to do in Farsite...

Summary of BFI Methodology

...but the overall methodology is, first, formally specify your system. Write a semantic spec, which describes the user's view of the system, and a distributed-system spec, which describes the designer's view of the system. Then write a refinement, how to interpret the distributed-system spec in semantic terms. Once you've got your specs, you modify the distributed-system spec to expresses the behavior of Byzantine-faulty components, which will break the refinement. Then, you simultaneously weaken the semantic spec, as little as you possibly can, to describe how faults can semantically manifest. And you improve the distributed-system spec to quarantine Byzantine faults, as best you can. Once you're again able to refine the distributed-system spec to the semantic spec, you know you're done.

Conclusions

In conclusion, BFT groups present a wonderful fault abstraction, but their throughput drops with increasing system scale. You can build a scalable system out of multiple BFT groups, but as system scale grows, there's an increasing probability that at least one group somewhere in the system will have more faulty machines than it can handle. If Byzantine faults are not quarantined by some mechanism, a single faulty group can corrupt the entire system. We've introduced Byzantine fault isolation, a methodology that uses formal specification to improve fault tolerance without hurting system throughput, unlike – dramatically unlike – increasing BFT group size.

Conclusions

Questions?