CIS 700/003: Distributed Systems
meet Social Networks

Byzantine behavior

February 18, 2010
Where are we?

- Measurement and analysis of MAD systems ✓
- Building MAD systems ✓
- Faults and Misbehavior
  - Rational behavior ✓
  - Byzantine behavior
  - Defenses
- Internet crime
- Privacy and confidentiality
- Novel opportunities
- Experience
My discussion points

- Is BFT sufficient to build a trustworthy service?
- Are the assumptions realistic? Are the guarantees strong enough?
- Overheads
- How general is this? Can it be applied to all systems?
- How common are Byzantine faults?
- How widely do you think this technique is used?
- How does this compare to software verification?
- "We omit discussion of how nodes recover from faults"
- Are there classes of faults BFT does not help against?
- Relation to FLP? Theory and practice?
- Fairness of their evaluation?
- Performance with vs. without faults
- Relationship to previous work
- Can we really rule out correlated faults? Do we have to?
Some of your discussion points

- NFS as the only application to evaluate?
- Low overhead
- Old hat - we've seen this again and again
- Relation to FLP
- Formalization
- Recovering faulty machines
- View change not explained; seems difficult
- Expensive voting procedure
- 'Frivolous' votes
- Can't you do it with 2f+1 nodes?
Papers related to BFT

- PBFT (OSDI 1999)
- Q/U (SOSP 2005)
- BAR (SOSP 2005)
- HQ (OSDI 2006)
- BFT2F (NSDI 2007)
- Zyzzyva (SOSP 2007)
- Shepherd (SOSP 2007)
- A2M (SOSP 2007)
- Aardvark (NSDI 2009)
- UpRight (SOSP 2009)

Not a complete list!
An almost magical guarantee

- At a very high level, a BFT-replicated service appears correct to clients even if some fraction of the replicas fail or domains misbehave in some (almost) arbitrary way
  - Wow!
Recap: The Byzantine model

- Assumption: Faulty nodes can behave arbitrarily
  - Subsumes all the other models we've discussed so far
  - Examples: Faulty nodes can crash, deviate from the protocol, act selfishly, lie, equivocate, attack other nodes, ...
  - Usually it is assumed that the faulty nodes still have limited capabilities (cannot suddenly factor large numbers or cut network links between other nodes)
Recap: The Byzantine model

- What does this buy us?
  - Extremely high resilience
  - Fewer assumptions → Fewer chances to get them wrong
  - Better robustness against intentional attacks

- This comes at a price, however
  - High complexity of the BFT protocol itself
  - High cost: Four machines (or more) to do the job of one
  - Message complexity; propagation delays; determinism
  - May have to rewrite the software from scratch
  - Need to ensure that no more than 33% of the nodes can be faulty at the same time
    - How do we do that?
Some examples of Byzantine faults

LAX Meltdown Caused By A Single Network Interface Card
By Meg Maroo on August 15, 2007 3:49 PM

According to the LA Times, the LAX computer meltdown that stranded 20,000 international passengers was the work of a single malfunctioning network interface card on a single desktop computer in the LAX international terminal. From the LA Times:

"The card, which allows computers to connect to a local area network, experienced a partial failure that started about 12:50 p.m. Saturday, slowing down the system, said Jennifer Connors, a chief in the office of field operations for the Customs and Border Protection agency.

As data overloaded the system, a domino effect occurred with other computer network cards, eventually causing a total system failure a little after 2 p.m., Connors said.

"All indications are there was no hacking, no tampering, no terrorist link, nothing like that," she said. "It was an internal problem" contained to the Los Angeles International Airport system."
Some examples of Byzantine faults

Amazon S3 Availability Event: July 20, 2008

We wanted to provide some additional detail about the problem we experienced on Sunday, July 20th.

At 8:40am PDT, error rates in all Amazon S3 datacenters began to quickly climb and our alarms went off. By 8:50am PDT, error rates were significantly elevated and very few requests were completing successfully. By 8:55am PDT, we had multiple engineers engaged and investigating the issue. Our alarms pointed at problems processing customer requests in multiple places within the system and across multiple data centers. While we began investigating several possible causes, we tried to restore system health by taking several actions to reduce system load. We reduced system load in several stages, but it had no impact on restoring system health.

At 9:41am PDT, we determined that servers within Amazon S3 were having problems communicating with each other. As background information, Amazon S3 uses a gossip protocol to quickly spread server state information throughout the system. This allows Amazon S3 to quickly route around failed or unreachable servers, among other things. When one server connects to another as part of processing a customer’s request, it starts by gossiping about the system state. Only after gossip is completed will the server send along the information related to the customer request. On Sunday, we saw a large number of servers that were spending almost all of their time gossiping and a disproportionate amount of servers that had failed while gossiping. With a large number of servers gossiping and failing while gossiping, Amazon S3 wasn’t able to successfully process many customer requests.

At 10:32am PDT, after exploring several options, we determined that we needed to shut down all communication between Amazon S3 servers, shut down all components used for request processing, clear the system’s state, and then reactivate the request processing components. By 11:05am PDT, all server-to-server communication was stopped, request processing components shut down, and the system’s state cleared. By 2:20pm PDT, we’d restored internal communication between all Amazon S3 servers and began reactivating request processing components concurrently in both the US and EU.

At 2:57pm PDT, Amazon S3’s EU location began successfully completing customer requests. The EU location came back online before the US because there are fewer servers in the EU. By 3:10pm PDT, request rates and error rates in the EU had returned to normal. At 4:02pm PDT, Amazon S3’s US location began successfully completing customer requests, and request rates and error rates had returned to normal by 4:58pm PDT.

http://status.aws.amazon.com/s3-20080720.html
Some examples of Byzantine faults

Ma.gnolia Suffers Major Data Loss, Site Taken Offline
By Michael Calore  January 30, 2009 | 12:56 pm | Categories: Uncategorized

There was a meltdown at bookmark sharing website Ma.gnolia Friday morning. The service lost both its primary store of user data, as well as its backup. The site has been taken offline while the team tries to reconstruct its databases, though some users may never see their stored bookmarks again.

The failure appears to be catastrophic. The company can’t say to what extent it will be able to restore any of its users’ data. It also says the data failure was so extensive, repairing the loss will take "days, not hours."

In light of today’s outage, many are questioning the reliability of web apps and web-based storage in general. Twitter in particular is full of users venting their suspicions.

"Cloud computing becomes fog when it goes down," says Todd Spragins in a Twitter post.

http://www.wired.com/epicenter/2009/01/magnolia-suffer/
Some examples of Byzantine faults

On July 2, from 6:45 AM PDT until 12:35 PM PDT, Google App Engine (App Engine) experienced an outage that ranged from partial to complete. Following is a timeline of events, an analysis of the technology and process failures, and a set of steps the team is committed to taking to prevent such an outage from happening again.

The App Engine outage was due to complete unavailability of the datacenter’s persistence layer, GFS, for approximately three hours. The GFS failure was abrupt for reasons described below, and as a consequence the data belonging to App Engine applications remained resident on GFS servers and was unreachable during this period. Since needed application data was completely unreachable for a longer than expected time period, we could not follow the usual procedure of serving of App Engine applications from an alternate datacenter, because doing so would have resulted in inconsistent or unavailable data for applications.

The root cause of the outage was a bug in the GFS Master server caused by another client in the datacenter sending it an improperly formed filehandle which had not been safely sanitized on the server side, and thus caused a stack overflow on the Master when processed.

http://groups.google.com/group/google-appengine/msg/ba95ded980c8c179
BFT in practice

- Very few real systems actually use it. Why?
- Getting BFT to work is hard
  - Prevent correlated faults: Different hardware, software, …
  - Correct configuration: Install all the cryptographic keys, etc.
  - Correct implementation: BFT protocols are highly complex
- But Byzantine faults are relatively rare
- Investigated faults in Yahoo!'s ZooKeeper service, which had been running for >1 year
  - 9 problems overall; 5 due to misconfiguration, 2 due to application bugs, 2 to server bugs
  - None would have been tolerated by BFT

Is ZooKeeper a good example to study?

Song, Junqueira, Reed: "BFT for the skeptics", BFTW3 workshop, Elche, 2009
BAR Fault Tolerance

- Observation: Nodes can depart from the protocol for different reasons
  - Broken nodes: Hardware fault, malicious operator, break-in...
  - Selfish nodes: Want to increase their utility
  - In BFT, the selfish nodes would be counted as Byzantine

- Idea: Model more than one class of nodes
  - Altruistic: Never depart from protocol
  - Selfish: Will depart from protocol if it benefits them
  - Byzantine: May behave arbitrarily, independent of benefit

- Approach: Carefully construct a BFT protocol such that it is in each node's best interest to follow it
  - Result: Safety and liveness as long as <1/3 of the nodes are Byzantine; the other nodes can all be rational

Aiyer et al.: "BAR Fault Tolerance for Cooperative Services", SOSP 2005
Take-away points

■ Byzantine fault model is extremely conservative
  ■ (Almost) no assumptions about faulty machines
  ■ Subsumes all the fault models we’ve discussed so far

■ It is possible to build systems that are resilient even to Byzantine faults
  ■ Can use BFT to maintain availability and integrity even if slightly less than a third of the machines are Byzantine
  ■ Wow!

■ But BFT is not perfect
  ■ Not effective against certain attacks, e.g., on confidentiality
  ■ Comes at a high price (resources, complexity)
  ■ Must ensure that \( \geq 2/3 \) of the replicas are correct at all times
What does this mean for MAD systems?

- Suppose you want to design a new, reliable MAD system

- Step #1: Pick a fault model (types of faults and misbehavior you expect)

- Step #2: Pick appropriate defenses
  - Example: Provide incentives to selfish participants, implement BFT/BAR to mask a certain number of Byzantine faults, etc.
  - We will look at more techniques later

© 2010 Andreas Haeberlen
Where are we?

- Measurement and analysis of MAD systems ✓
- Building MAD systems ✓
- Faults and Misbehavior
  - Rational behavior ✓
  - Byzantine behavior
  - Defenses → Next time
- Internet crime
- Privacy and confidentiality
- Novel opportunities
- Experience
Stay tuned!

Next week you will learn:

How to use reputations to build better MAD distributed systems