CS162
Operating Systems and Systems Programming
Lecture 23

Distributed Systems

April 15, 2010

Benjamin Hindman
http://inst.eecs.berkeley.edu/~cs162

Distributed Systems are Everywhere!

- We need (want?) to share physical devices (e.g., printers) and information (e.g., files)
- Many applications are distributed in nature (e.g., ATM machines, airline reservations)
- Many large problems can be solved by decomposing into lots of smaller problems that can be run in parallel (e.g., MapReduce, SETI@home)

What makes building distributed systems interesting?

- Programming models
- Transparency
- Fault-tolerance
- Performance
- Scalability
- Consistency
- Security

Distributed Applications

- How do you actually program a distributed application?

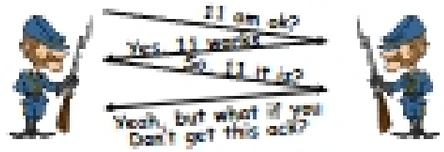
- Use networking building blocks to provide a basic send/receive abstraction (message passing)
  - Semantics: sender asks a specific receiver and receiver gets all or none of the message
  - Queue incoming messages on receive side

Using Messages: Send/Receive behavior

- When should send return?
  - Asynchronous: return immediately
  - Synchronous: return after …
    - Receiver acts message? (i.e., ack received)
    - When message is safely buffered on destination?
    - Right away, if message is buffered on source node?
- Main question here:
  - When can the sender be sure that receiver actually received the message?

General's Paradox

- General's paradox:
  - Constraints of problem:
    - Two generals, on separate mountains
    - Can only communicate via messengers
    - Messengers can be captured
  - Problem: need to coordinate attack
    - If they attack at different times, they all die
    - If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



Distributed Decision Making

- Why is distributed decision making desirable?
  - Fault Tolerance! A group of machines can come to a decision even if one or more of them fail during the process

Distributed Transactions

- Since we can't solve the General's Paradox, let's solve a related problem, distributed transaction: N machines agree to do something, or not do it, atomically
- Why should we care? Banks do this every day (every minute, every second, …)
- Two-Phase Commit Protocol
  - Phase 1, coordinator sends out a request to commit
    - each participant responds with yes or no
  - Phase 2
    - If everyone says yes, coordinator sends out a commit
    - If someone says no, coordinator sends out an abort

Two-Phase Commit Details

- Each participant uses a local, persistent, corrupt-free log to keep track of whether a commit has happened
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Log can be used to complete this process such that all machines either commit or don't commit
  - Timeouts can be used to retry if coordinator doesn't hear from all participants

Two-Phase Commit Example

- Simple Example: A=WellsFargo, B=Chase
  - Phase 1:
    - A writes "Begin transaction" to log
    - A→B: OK to transfer funds to me?
    - B: Not enough funds
    - B→A: Transaction aborted: A writes "Abort" to log
    - A: Enough funds
    - B: Write new account balance & promise to commit to log
    - B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    - A: write new account balance to log
    - A: write "Commit" to log
    - A: send message to B that commit occurred: wait for ack
    - B: write "Commit" to log
- What if B crashes at beginning?
  - Wakes up, does nothing; A will timeout, abort and retry
- What if A crashes at beginning of phase 2?
  - Wakes up, sees that there is a transaction in progress; sends "Abort" to B
- What if B crashes at beginning of phase 2?
  - B comes back up, looks at log; when A sends it "Commit" message, it will say "oh, ok" commit"

Two-Phase Commit Gotchas

- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (Site A) and crashes
    - Site A crashes
    - Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to Site A asking what happened. At this point, B cannot decide to abort, because update may have committed
    - B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternatives such as "Three Phase Commit" don't have this blocking problem
- What happens if one or more of the participants is malicious?

Remote Procedure Call

- Raw messaging is a bit too low-level for programming
- Another option: Remote Procedure Call (RPC)
  - Looks like a local procedure call on client:

```
file.read(1024);
```

  - Translated automatically into a procedure call on remote machine (server)
- Implementation:
  - Uses request/response message passing "under the covers"