CS162
Operating Systems and
Systems Programming
Lecture 23

Distributed Decision Making (Con’t),
Networking and TCP/IP

April 19th, 2022
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Societal Scale Information Systems

- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them

Internet Connectivity

MEMS for Sensor Nets

Scalable, Reliable, Secure Services

Databases
Information Collection
Remote Storage
Online Games
Commerce

…
Centralized vs Distributed Systems

- **Centralized System**: major functions performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- **Distributed System**: physically separate computers working together on task
  - Early model: multiple servers working together
    - Probably in the same room or building
    - Often called a “cluster”
  - Later models: peer-to-peer/wide-spread collaboration
Distributed Systems: Motivation/Issues/Promise

• Why do we want distributed systems?
  – Cheaper and easier to build lots of simple computers
  – Easier to add power incrementally
  – Users can have complete control over some components
  – Collaboration: much easier for users to collaborate through network resources (such as network file systems)

• The promise of distributed systems:
  – Higher availability: one machine goes down, use another
  – Better durability: store data in multiple locations
  – More security: each piece easier to make secure
Distributed Systems: Reality

• Reality has been disappointing
  – **Worse availability**: depend on every machine being up
    » Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
  – **Worse reliability**: can lose data if any machine crashes
  – **Worse security**: anyone in world can break into system

• Coordination is more difficult
  – Must coordinate multiple copies of shared state information
  – What would be easy in a centralized system becomes a lot more difficult

• Trust/Security/Privacy/Denial of Service
  – Many new variants of problems arise as a result of distribution
  – Can you trust the other members of a distributed application enough to even perform a protocol correctly?
  – Corollary of Lamport’s quote: “A distributed system is one where you can’t do work because some computer you didn’t even know existed is successfully coordinating an attack on my system!”
Distributed Systems: Goals/Requirements

- **Transparency**: the ability of the system to mask its complexity behind a simple interface

  - **Possible transparencies:**
    - **Location**: Can’t tell where resources are located
    - **Migration**: Resources may move without the user knowing
    - **Replication**: Can’t tell how many copies of resource exist
    - **Concurrency**: Can’t tell how many users there are
    - **Parallelism**: System may speed up large jobs by splitting them into smaller pieces
    - **Fault Tolerance**: System may hide various things that go wrong

- Transparency and collaboration require some way for different processors to communicate with one another
A protocol is an agreement on how to communicate, including:
- **Syntax**: how a communication is specified & structured
  - Format, order messages are sent and received
- **Semantics**: what a communication means
  - Actions taken when transmitting, receiving, or when a timer expires

Described formally by a state machine:
- Often represented as a message transaction diagram
- Can be a partitioned state machine: two parties synchronizing duplicate sub-state machines between them
- Stability in the face of failures!
Examples of Protocols in Human Interactions

- **Telephone**
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. 
  6. Caller: “Hi, it’s Anthony…”
     Or: “Hi, it’s me” (← what’s *that* about?)
  7. Caller: “Hey, do you think … blah blah blah …” **pause**

  1. Callie: “Yeah, blah blah blah …” **pause**
  2. Caller: Bye
  3. 
  4. Hang up
Global Communication: The Problem

- Many different applications
  - email, web, P2P, etc.
- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?
  - Re-implement every application for every technology?
- No! But how does the Internet design avoid this?
Solution: Intermediate Layers

• Introduce intermediate layers that provide set of abstractions for various network functionality & technologies
  – A new app/media implemented only once
  – Variation on “add another level of indirection”

• Goal: Reliable communication channels on which to build distributed applications
There is just **one** network-layer protocol, IP. The “narrow waist” facilitates interoperability.
Implications of Hourglass

Single Internet-layer module (IP):

• Allows arbitrary networks to interoperate
  – Any network technology that supports IP can exchange packets

• Allows applications to function on all networks
  – Applications that can run on IP can use any network

• Supports simultaneous innovations above and below IP
  – But changing IP itself, i.e., IPv6, very involved
Drawbacks of Layering

• Layer N may duplicate layer N-1 functionality
  – E.g., error recovery to retransmit lost data
• Layers may need same information
  – E.g., timestamps, maximum transmission unit size
• Layering can hurt performance
  – E.g., hiding details about what is really going on
• Some layers are not always cleanly separated
  – Inter-layer dependencies for performance reasons
  – Some dependencies in standards (header checksums)
• Headers start to get really big
  – Sometimes header bytes >> actual content
End-To-End Argument

- Hugely influential paper: “End-to-End Arguments in System Design” by Saltzer, Reed, and Clark (‘84)
- “Sacred Text” of the Internet
  - Endless disputes about what it means
  - Everyone cites it as supporting their position
- Simple Message: Some types of network functionality can only be correctly implemented end-to-end
  - Reliability, security, etc.
- Because of this, end hosts:
  - Can satisfy the requirement without network’s help
  - Will/must do so, since can’t rely on network’s help
- Therefore don’t go out of your way to implement them in the network
Example: Reliable File Transfer

- Solution 1: make each step reliable, and then concatenate them

- Solution 2: end-to-end check and try again if necessary
Discussion

• Solution 1 is *incomplete*
  – What happens if memory is corrupted?
  – Receiver has to do the check anyway!

• Solution 2 is *complete*
  – Full functionality can be entirely implemented at application layer with **no** need for reliability from lower layers

• *Is there any need to implement reliability at lower layers?*
  – Well, it could be **more efficient**
End-to-End Principle

Implementing complex functionality in the network:
• Doesn’t reduce host implementation complexity
• Does increase network complexity
• Probably imposes delay and overhead on all applications, even if they don’t need functionality

• However, implementing in network can enhance performance in some cases
  – e.g., very lossy link
Conservative Interpretation of E2E

- Don't implement a function at the lower levels of the system unless it can be completely implemented at this level

- Or: Unless you can relieve the burden from hosts, don't bother
Moderate Interpretation

- Think twice before implementing functionality in the network
- If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement
- But do so only if it does not impose burden on applications that do not require that functionality
- This is the interpretation we are using

- Is this still valid?
  - What about Denial of Service?
  - What about Privacy against Intrusion?

  - Perhaps there are things that must be in the network???
Distributed Applications

• How do you actually program a distributed application?
  – Need to synchronize multiple threads, running on different machines
    » No shared memory, so cannot use test&set

  – One Abstraction: send/receive messages
    » Already atomic: no receiver gets portion of a message and two receivers cannot get same message

• Interface:
  – Mailbox (mbox): temporary holding area for messages
    » Includes both destination location and queue
  – Send(message, mbox)
    » Send message to remote mailbox identified by mbox
  – Receive(buffer, mbox)
    » Wait until mbox has message, copy into buffer, and return
    » If threads sleeping on this mbox, wake up one of them
Using Messages: Send/Receive behavior

• When should `send(message, mbox)` return?
  – When receiver gets message? (i.e. ack received)
  – When message is safely buffered on destination?
  – Right away, if message is buffered on source node?

• Actually two questions here:
  – When can the sender be sure that receiver actually received the message?
  – When can sender reuse the memory containing message?

• Mailbox provides 1-way communication from T1→T2
  – T1→buffer→T2
  – Very similar to producer/consumer
    » Send = V, Receive = P
    » However, can't tell if sender/receiver is local or not!
Messaging for Producer-Consumer Style

• Using send/receive for producer-consumer style:

  Producer:
  ```
  int msg1[1000];
  while(1) {
    prepare message;
    send(msg1,mbox);
  }
  ```

  Consumer:
  ```
  int buffer[1000];
  while(1) {
    receive(buffer,mbox);
    process message;
  }
  ```

• No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  – This is one of the roles of the window in TCP: window is size of buffer on far end
  – Restricts sender to forward only what will fit in buffer
Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - Read a file stored on a remote machine
    - Request a web page from a remote web server
  - Also called: client-server
    - Client = requester, Server = responder
    - Server provides “service” (file storage) to the client

- Example: File service

  Client: (requesting the file)
  ```
  char response[1000];
  send("read rutabaga", server_mbox);
  receive(response, client_mbox);
  ```

  Server: (responding with the file)
  ```
  char command[1000], answer[1000];
  receive(command, server_mbox);
  decode command;
  read file into answer;
  send(answer, client_mbox);
  ```
Distributed Consensus Making

• Consensus problem
  – All nodes propose a value
  – Some nodes might crash and stop responding
  – Eventually, all remaining nodes decide on the same value from set of proposed values

• Distributed Decision Making
  – Choose between “true” and “false”
  – Or Choose between “commit” and “abort”

• Equally important (but often forgotten!): make it durable!
  – How do we make sure that decisions cannot be forgotten?
    » This is the “D” of “ACID” in a regular database
  – In a global-scale system?
    » What about erasure coding or massive replication?
    » Like BlockChain applications!
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
General’s Paradox (con’t)

• Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  – Remarkably, “no”, even if all messages get through
    – No way to be sure last message gets through!
    – In real life, use radio for simultaneous (out of band) communication
  
• So, clearly, we need something other than simultaneity!
Two-Phase Commit

• Since we can’t solve the General’s Paradox (i.e. simultaneous action), let’s solve a related problem

• Distributed transaction: Two or more machines agree to do something, or not do it, atomically
  – No constraints on time, just that it will eventually happen!

• Two-Phase Commit protocol: Developed by Turing award winner Jim Gray
  – (first Berkeley CS PhD, 1969)
  – Many important Database breakthroughs also from Jim Gray
Two-Phase Commit Protocol

- Persistent stable log on each machine: keep track of whether 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
- Prepare Phase:
  - The global coordinator requests that all participants will promise to commit or rollback the transaction
  - Participants record promise in log, then acknowledge
  - If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log
- Commit Phase:
  - After all participants respond that they are prepared, then the coordinator writes "Commit" to its log
  - Then asks all nodes to commit; they respond with ACK
  - After receive ACKs, coordinator writes "Got Commit" to log
- Log used to guarantee that all machines either commit or don't
2PC Algorithm

- One coordinator
- N workers (replicas)

High level algorithm description:
  - Coordinator asks all workers if they can commit
  - If all workers reply “VOTE-COMMIT”, then coordinator broadcasts “GLOBAL-COMMIT”
    Otherwise coordinator broadcasts “GLOBAL-ABORT”
  - Workers obey the GLOBAL messages

- Use a persistent, stable log on each machine to keep track of what you are doing
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
Two-Phase Commit: Setup

- One machine (*coordinator*) initiates the protocol
- It asks *every* machine to **vote** on transaction

- Two possible votes:
  - **Commit**
  - **Abort**

- Commit transaction only if unanimous approval
Two-Phase Commit: Preparing

Worker Agrees to Commit
• Machine has **guaranteed** that it will accept transaction
• Must be **recorded in log** so machine will remember this decision if it fails and restarts

Worker Agrees to Abort
• Machine has **guaranteed** that it will **never accept** this transaction
• Must be **recorded in log** so machine will remember this decision if it fails and restarts
Two-Phase Commit: Finishing

Commit Transaction
- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction
- Coordinator learns *at least one machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters
Two-Phase Commit: Finishing

Commit Transaction
- Coordinator learns all machines have agreed to commit
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction
- Coordinator learns at least one machine has voted to abort
- Record decision to abort in local log
- Do not apply transaction, inform voters

Because no machine can take back its decision, exactly one of these will happen
Administrivia

- Midterm 3: Thursday 4/28: 7-9PM
  - All course material
  - Review session 4/25
Coordinator Algorithm

- Coordinator sends VOTE-REQ to all workers

- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If don’t receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for VOTE-REQ from coordinator
- If ready, send VOTE-COMMIT to coordinator
- If not ready, send VOTE-ABORT to coordinator
  - And immediately abort

- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort
Failure Free Example Execution

coordinator

worker 1

worker 2

worker 3

VOTE-REQ

GLOBAL-COMMIT

VOTE-COMMIT

time
State Machine of Coordinator

- Coordinator implements simple state machine:

```
  INIT
  |--- Recv: START
  |     Send: VOTE-REQ
  |
  WAIT
  |--- Till all votes
  |     Send: GLOBAL-ABORT
  |
  |--- Till all VOTE-COMMIT
  |     Send: GLOBAL-COMMIT
  |
  ABORT
  COMMIT
```
State Machine of Workers

- INIT
  - Recv: VOTE-REQ
  - Send: VOTE-ABORT
  - Recv: VOTE-REQ
  - Send: VOTE-COMMIT

- READY
  - Recv: GLOBAL-ABORT
  - Recv: GLOBAL-COMMIT

- ABORT
  - Commits

- COMMIT
  - Commits
Dealing with Worker Failures

- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in “WAIT” state
- In WAIT, if doesn’t receive N votes, it times out and sends GLOBAL-ABORT
Example of Worker Failure

INIT

WAIT

ABORT

COMM

GLOBAL

ABORT

timeout

VOTE-REQ

VOTE-COMMIT

coordinator

worker 1

worker 2

worker 3

time
Dealing with Coordinator Failure

- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
  - If coordinator fails, workers must BLOCK waiting for coordinator to recover and send GLOBAL_-* message
Example of Coordinator Failure #1

- Coordinator
  - INIT
  - READY
  - ABORT
  - COMM

- Worker 1
- Worker 2
- Worker 3

Processes:
- VOTE-REQ
- VOTE-ABORT
- Timeout
Example of Coordinator Failure #2

VOTE-REQ

VOTE-COMMIT

GLOBAL-ABORT

block waiting for coordinator

coordinator

worker 1

worker 2

worker 3

restarted

INIT

READY

ABORT

COMM
Durability

• All nodes use stable storage to store current state
  – stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
  – E.g.: SSD, NVRAM

• Upon recovery, nodes can restore state and resume:
  – Coordinator aborts in INIT, WAIT, or ABORT
  – Coordinator commits in COMMIT
  – Worker aborts in INIT, ABORT
  – Worker commits in COMMIT
  – Worker “asks” Coordinator in READY