Recall: Societal Scale Information Systems

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Networks form systems
  - Collaboration requires users to collaborate through network resources (such as information services)
  - Users can have complete control over some components
  - Easier to add power incrementally
  - Easier and faster to build lots of simple components

Centralized vs Distributed Systems

Centralized System: major function performed by a single physical computer

Distributed System: physically separate computers working together on task

The promise of distributed systems:

- Better security: each piece easier to make secure
- Better reliability: fewer single points of failure
- Higher availability: one machine goes down, use another
- Lower cost: each computer costs less

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)

Centralized System

Peer-to-Peer Model

Client/Server Model

Distributed Systems: Motivation/Issues/Promise

Why do we want distributed systems?

- Cheaper and easier to build lots of simple components
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration is much easier for users to collaborate through network resources (such as information services)
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 the world can break into your 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 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 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

How do entities communicate? A Protocol!
- A protocol is an agreement on how to communicate, including:
  - Syntax: how a communication is specified and 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

Examples of Protocols in Human Interactions
- Telephone
  1. (Pick up the phone)
  2. Listen for a dial tone / see you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: "Hello?"
  6. Caller: "Hi, it’s Anthony….
  7. Callee: "Yeah, blah blah…"
  8. Caller: "Bye"
- Or:
  5. Callee: "Hello?"
  6. Caller: "Hi, it’s me (what’s that about?)"
  7. Callee: "Yeah, blah blah…"
  8. Caller: "Bye"

Protocol Exchange
- How does communication look like?
  - Two parties synchronizing duplicate sub-state machines

Distributed Systems: Reality
- Reality has been disappointing
  - Depend on every machine being up
  - Can lose data if any machine crashes
  - Anyone in the world can break into your system
  - Coordination is 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 perform a protocol correctly?
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?
  – 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 functionalities
  – A new app/media implemented only once
  – Variations on "add another level of indirection"
• Goal: Reliable communication channels on which to build distributed applications

The "narrow waist" facilitates interoperability

There is just one network-layer protocol: IP.

The Internet Hourglass

There is just one network-layer protocol: IP.

The Internet Hourglass Model

There is just one network-layer protocol: IP.

Implications of Hourglass

Single Internet-layer module (IP):

- Allows arbitrary networks to interoperate
  – Any network technology that supports IP can exchange packets
- Allows arbitrary networks to interoperate
- Supports simultaneous innovations above and below IP
- Application protocols can run on IP that can use any network
- Any network technology that supports IP can exchange packets
- Single Internet-layer module (IP)

Cost: Perils of communication channels on which to build distributed applications

- Losses on each intermediate layer
- Any application implementing only once
- Introduce intermediate layers that provide set of abstractions for various networks
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
    – Headers start to get really big
    – Some dependences in standards (header lengths)
    – Inter-layer dependencies for performance reasons
    – Some layers are not always cleanly separated

End-To-End Argument

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 network's help
    – Can satisfy the requirement without network's help
  – Reliability, security, etc.

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

OS Appl.

Host A

Host B

OK

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
  – Reliability is considered

• Is there any need to implement reliability at lower layers?
  – Well, it could be more efficient
Implementing complex functionality in the network:
- Does not 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
  - This is the interpretation we are using
  - Perhaps there are things that must be in the network...
- Does the E2E interpretation make sense in the following scenario?
  - Hosts do some computations, and need to output data in a sequence:
- 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:
- Send/receive messages
  - 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

End-to-End Principle
- Implementing complex functionality in the network:
  - Does not 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
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 sends message, T2 receives
– Very similar to producer/consumer
  » Send = V, Receive = P
» However, cannot tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

• Using send/receive for producer-consumer style:
  Producer:
  ```c
  int msg1[1000];
  while(1) {
    prepare message;
    send(msg1, mbox);
  }
  ```
  Consumer:
  ```c
  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: requestor, Server: responder
    » Server provides "service" (file storage) to the client

Example: File service
Client: (requesting the file)
```c
char response[1000];
send("read rutabaga", server_mbox);
receive(response, client_mbox);
``` Server: (responding with the file)
```c
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, remaining nodes decide on the same value from proposed values
  – How do we make sure that decision cannot be forgotten?
• Distributed Decision Making
  – Choose between "true" and "false"
  – Or Choose between "commit" and "abort"
  – Equally important (but often forgotten!): make it durable!
    » 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!

Using Messages: Send/Receive behavior

```c
send("hello", client_mbox);
read the incoming messages
read (sensitive) data from the file
```
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.

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 it crashed.

**Prepare Phase**
- Global coordinator requests all participants 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"
- After receive ACK, coordinator writes "Commit" to log
- Then ask all nodes to commit they responded with ACK to is log

- Log used to guarantee that all messages either commit or don't.
- After receive ACK, coordinator writes "Commit" to log
- Then ask all nodes to commit they responded with ACK to is log
- After all participants respond that they are prepared, then the coordinator writes "Commit"

- Commutative property in log
- All copies of log are identical
- All nodes that received a message get notified
- Can messengers pass messages that cause the network to converge to guarantee two entities do it
Two-Phase Commit 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 the transaction
- If a majority of machines vote to commit, the coordinator learns all machines have voted to commit
- Otherwise, the coordinator learns at least one machine has voted to abort

Two-Phase Commit: Preparing
- Worker agrees to commit
  - Machine has guaranteed that it will accept this transaction
  - The decision is recorded in the log so the machine will remember this decision if it fails and restarts
- Worker agrees to abort
  - Machine has guaranteed that it will never accept this transaction
  - The decision is recorded in the log so the machine will remember this decision if it fails and restarts

Two-Phase Commit: Finishing
- Commit transaction
  - Coordinator learns all machines have agreed to commit
  - This decision is recorded in the local log
  - Apply the transaction
- Abort transaction
  - Coordinator learns at least one machine has voted to abort
  - This decision is recorded in the local log
  - Do not apply the transaction
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
- Review session 4/25
- All course material
- Final exam: Monday 5/2, 7:45AM
State Machine of Coordinator

- Coordinator implements simple state machine:
  - INIT
  - WAIT
  - ABORT
  - COMMIT

  - Recv: START
  - Send: VOTE

  - Recv: VOTE
  - Send: VOTE

- In "WAIT" state, Coordinator only waits for votes. If it doesn't receive N votes, it times out and sends GLOBAL-ABORT.

  - Recv: all VOTE
  - Send: GLOBAL

State Machine of Worker

- Worker also implements simple state machine:
  - INIT
  - READY
  - ABORT
  - COMMIT

  - Recv: VOTE
  - Send: VOTE

  - Recv: VOTE
  - Send: VOTE

Dealing with Worker Failures

- Failure only affects states in which the Coordinator is waiting for messages.

- In "WAIT" state, if doesn't receive N votes, it times out and sends GLOBAL-ABORT.

Example of Worker Failure

- Coordinator sends GLOBAL-ABORT.
- Worker 1 times out and sends GLOBAL-ABORT.
- Worker 2 and Worker 3 vote COMMIT.
- Coordinator receives VOTE and sends GLOBAL-ABORT.
- Coordinator receives VOTE and sends GLOBAL-ABORT.
- Coordinator receives VOTE and sends GLOBAL-ABORT.
- Coordinator times out and sends GLOBAL-ABORT.
Dealing with Coordinator Failure

- Worker waits for VOTE-REQ in INIT – Worker can timeout 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

 timeout

 abort

 vote

 timeout

 abort

 vote

 coordinator

 Example of Coordinator Failure #2

 coordinator

 abort

 ready

 init

 abort

 global

 commit

 vote

 ready

 vote

 commit

 abort

 global

 Example with Coordinator Failure

 To recover and send GLOBAL-* message, all coordination fails. Workers must BLOCK waiting for coordinator. Worker waits for GLOBAL-* message in READY – Worker can timeout and abort (coordination fails)

 Worker waits for VOTE-REQ in INIT

 cowardordinator

 global abort

 vote ready

 abort
c

c

c

c

c

c

c

c

c

Durability

- All nodes use stable storage to store current state

- All nodes use stable storage to store current state

- All nodes use stable storage to store current state

- All nodes use stable storage to store current state

- All nodes use stable storage to store current state