Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Centralized vs Distributed Systems

**Centralized System:** System in which major functions are performed by a single physical computer
- Originally, everything on single computer
- Later: client/server model

**Distributed System:** physically separate computers working together on some 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

• 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

• Reality has been disappointing
  – Worse availability: depend on every machine being up
    » Lamport: “a distributed system is one where I can’t do work because some machine I’ve never heard of isn’t working!”
  – 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 (using only a network)
  – What would be easy in a centralized system becomes a lot more difficult
Recall: 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 in the system

- **Transparency and collaboration require some way for different processors to communicate with one another**
Networking Definitions

- **Network**: physical connection that allows two computers to communicate
- **Packet**: unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- **Protocol**: agreement between two parties as to how information is to be transmitted
Recall: Use of TCP: Sockets

• **Socket:** an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    » Same interface regardless of location of other end
    » Could be local machine (called “UNIX socket”) or remote machine (called “network socket”)
  - First introduced in 4.2 BSD UNIX: big innovation at time
    » Now most operating systems provide some notion of socket

• **Using Sockets for Client-Server (C/C++ interface):**
  - On server: set up “server-socket”
    » Create socket, Bind to protocol (TCP), local address, port
    » Call listen(): tells server socket to accept incoming requests
    » Perform multiple accept() calls on socket to accept incoming connection request
    » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    » Create socket, Bind to protocol (TCP), remote address, port
    » Perform connect() on socket to make connection
    » If connect() successful, have socket connected to server
Recall: Socket Setup over TCP/IP

- **Server Socket**: Listens for new connections
  - Produces new sockets for each unique connection

- **Things to remember**:
  - **Connection involves 5 values**:
    [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port “randomly” assigned
    » Done by OS during client socket setup
  - Server Port often “well known”
    » 80 (web), 443 (secure web), 25 (sendmail), etc
    » Well-known ports from 0—1023
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 \texttt{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
    - However, can't 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
  - TCP manages the 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);
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
  - No way to be sure last message gets through!

- No way to be sure last message gets through!

11 am ok?
Yes, 11 works
So, 11 it is?
Yeah, but what if you Don't get this ack?

Yes, 11 works
So, 11 it is?
Yeah, but what if you Don't get this ack?

- No way to be sure last message gets through!
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 machines agree to do something, or not do it, atomically
- 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 can be used to complete this process such that all machines either commit or don't commit
2PC Algorithm

- Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
- 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
**Coordinator Algorithm**

- Coordinator sends **VOTE-REQ** to all workers
- 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 **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If doesn’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

time
Coordinator implements simple state machine:

- **INIT**
  - Recv: START
  - Send: VOTE-REQ

- **WAIT**
  - Recv: VOTE-ABORT
  - Send: GLOBAL-ABORT
  - Recv: all VOTE-COMMIT
  - Send: GLOBAL-COMMIT

- **ABORT**
- **COMMIT**
State Machine of Workers

- **INIT**
  - Receive: VOTE-REQ
  - Send: VOTE-ABORT
  - Receive: VOTE-REQ
  - Send: VOTE-COMMIT

- **READY**
  - Receive: VOTE-REQ
  - Send: VOTE-COMMIT
  - Receive: GLOBAL-ABORT
  - Receive: GLOBAL-COMMIT

- **ABORT**

- **COMMIT**
Dealing with Worker Failures

• How to deal with worker failures?
  - Failure only affects states in which the node 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

Coordinator

Worker 1

Worker 2

Worker 3

INIT

WAIT

ABORT

COMM

timeout

GLOBAL-ABORT

VOTE-REQ

VOTE-COMMIT

time

X

timeout

GLOBAL-ABORT

VOTE-REQ

VOTE-COMMIT

VOTE-REQ
Dealing with Coordinator Failure

- How to deal with coordinator failures?
  - 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

Init

- Recv: VOTE-REQ
  - Send: VOTE-ABORT

- Recv: VOTE-REQ
  - Send: VOTE-COMMIT

Ready

- Recv: GLOBAL-ABORT

Aborts

- Recv: GLOBAL-COMMIT

Commits
Example of Coordinator Failure #1

Coordinator

Worker 1

Worker 2

Worker 3

INIT

READY

ABORT

COMM

VOTE-REQ

timeout

timeout

timeout

VOTE-ABORT
Example of Coordinator Failure #2

INIT

READY

ABORT
COMM

coordinator

worker 1
VOTE-REQ

worker 2
VOTE-COMMIT

worker 3

restarted

block waiting for coordinator

GLOBAL-ABORT
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.

- Upon recovery, it 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
Blocking for Coordinator to Recover

- A worker waiting for global decision can ask fellow workers about their state
  - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
    » Thus, worker can safely abort or commit, respectively
  - If another worker is still in INIT state then both workers can decide to abort
  - If all workers are in ready, need to BLOCK (don’t know if coordinator wanted to abort or commit)
Distributed Decision Making Discussion

• 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
  - After decision made, result recorded in multiple places

• 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

• PAXOS: An alternative used by GOOGLE and others that does not have this blocking problem

• What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making
Byzantine General’s Problem

• Byzantine General’s Problem (n players):
  – One General
  – n−1 Lieutenants
  – Some number of these (f) can be insane or malicious
• The commanding general must send an order to his n−1 lieutenants such that:
  – All loyal lieutenants obey the same order
  – If the commanding general is loyal, then all loyal lieutenants obey the order he sends
Byzantine General's Problem (con't)

• Impossibility Results:
  - Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things
  - With f faults, need n > 3f to solve problem

• Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity $O(n^2)$
    » One from MIT, for instance (Castro and Liskov, 1999)

• Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them (< n/3 ) are malicious
Remote Procedure Call

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    ```
    remoteFileSystem→Read(“rutabaga”);
    ```
  - Translated automatically into call on server:
    ```
    fileSys→Read(“rutabaga”);
    ```
- Implementation:
  - Request-response message passing (under covers!)
  - “Stub” provides glue on client/server
    - Client stub is responsible for “marshalling” arguments and “unmarshalling” the return values
    - Server-side stub is responsible for “unmarshalling” arguments and “marshalling” the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.
RPC Information Flow

- **Client (caller)**
  - Call
  - Return

- **Server (callee)**
  - Call
  - Return

- **Packet Handler**
  - Send
  - Receive

- **Client Stub**
  - Bundle args
  - Unbundle return values

- **Server Stub**
  - Unbundle arguments

- **Network**
  - Mbox1
  - Mbox2

**Machine A**

**Machine B**
RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)

- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an “interface definition language (IDL)”
    » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    » Code for server to unpack message, call procedure, pack results, send them off

- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    » Convert everything to/from some canonical form
    » Tag every item with an indication of how it is encoded (avoids unnecessary conversions).
RPC Details (continued)

• How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint
    (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a
    network endpoint
    » This is another word for “naming” at network level
    » Static: fixed at compile time
    » Dynamic: performed at runtime

• Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    » Name service provides dynamic translation of service \( \rightarrow \) mbox
  - Why dynamic binding?
    » Access control: check who is permitted to access service
    » Fail-over: If server fails, use a different one

• What if there are multiple servers?
  - Could give flexibility at binding time
    » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    » Choose unloaded server for each new request
    » Only works if no state carried from one call to next

• What if multiple clients?
  - Pass pointer to client-specific return mbox in request
Problems with RPC

- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    » User-level bug causes address space to crash
    » Machine failure, kernel bug causes all processes on same machine to fail
    » Some machine is compromised by malicious party
- Before RPC: whole system would crash/die
- After RPC: One machine crashes/compromised while others keep working
- Can easily result in inconsistent view of the world
  » Did my cached data get written back or not?
  » Did server do what I requested or not?
- Answer? Distributed transactions/Byzantine Commit

- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    » Caching can help, but may make failure handling complex
Cross-Domain Communication/Location Transparency

• How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc...
  - File System
  - Pipes (1-way communication)
  - “Remote” procedure call (2-way communication)

• RPC's can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same

• Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)
Microkernel operating systems

• Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine

  ![Diagram of monolithic and microkernel structures]
  
  **Monolithic Structure**

  **Microkernel Structure**

• Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    » For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.
Summary

• Protocol: Agreement between two parties as to how information is to be transmitted

• Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit

• Byzantine General’s Problem: distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often “f” of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if $n \geq 3f+1$

• Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)