

CS162  
Operating Systems and  
Systems Programming  
Lecture 24

Distributed 1: End-To-End Argument,  
Distributed Decision Making, 2PC,  
Remote Procedure Calls (RPC)

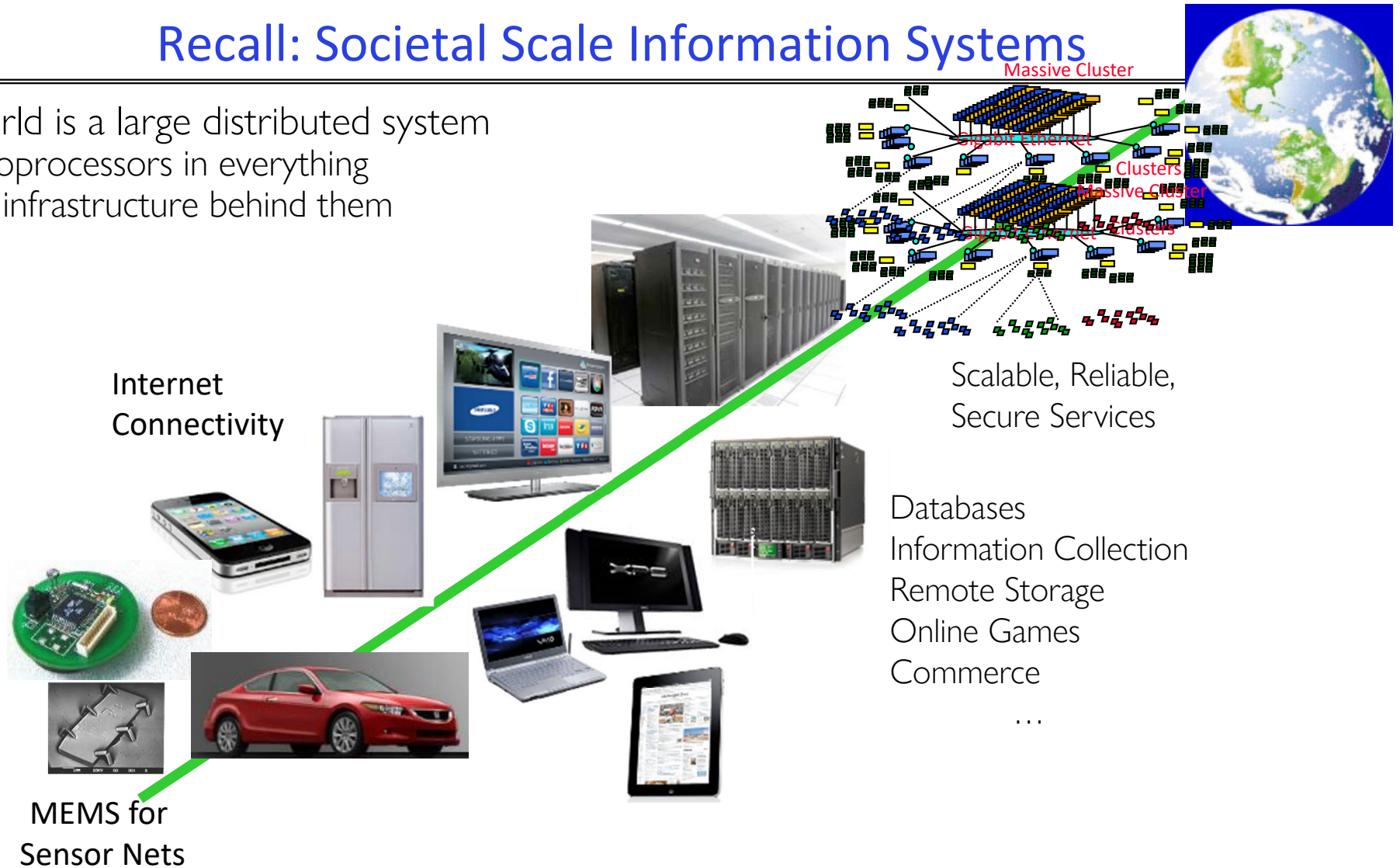
April 23<sup>rd</sup>, 2026

Prof. John Kubiatowicz

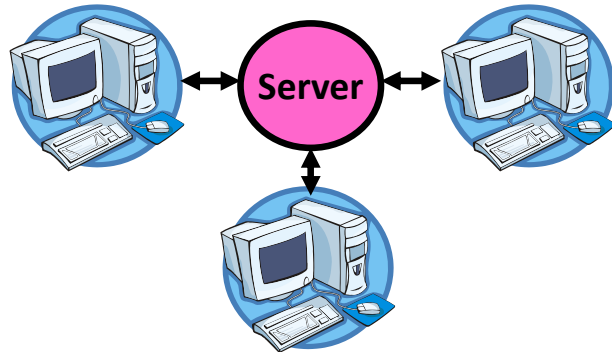
<http://cs162.eecs.Berkeley.edu>

# Recall: Societal Scale Information Systems

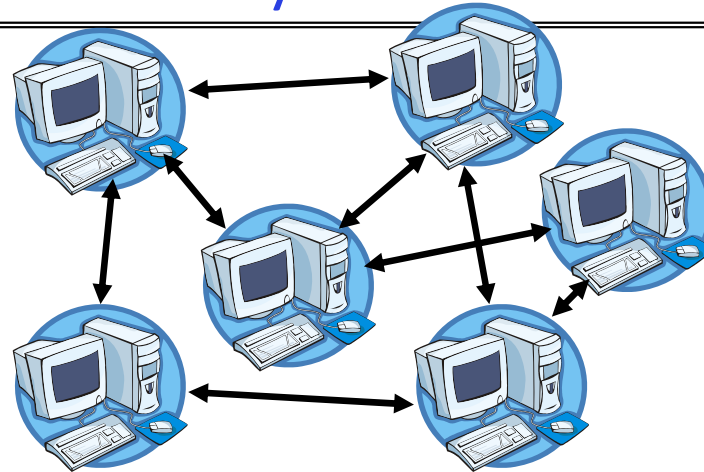
- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them



## Centralized vs Distributed Systems



**Client/Server Model**



**Peer-to-Peer Model**

- **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

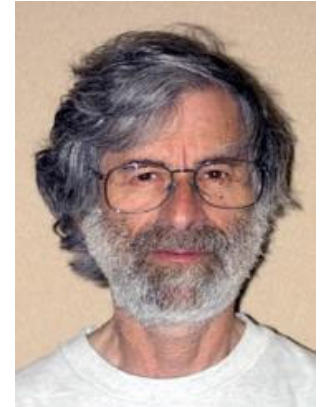
---

- 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!”

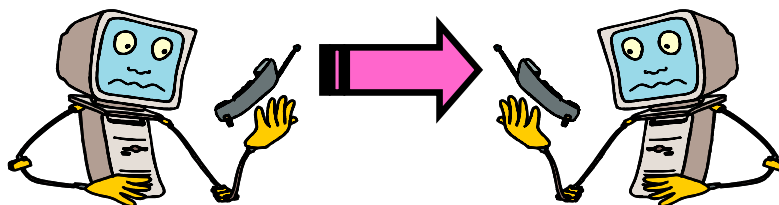


**Leslie Lamport**

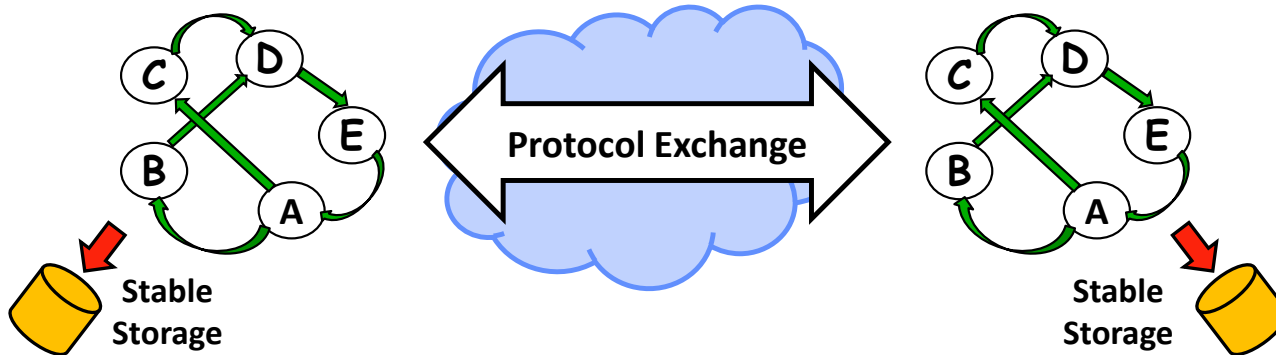
## Distributed Systems: Goals/Requirements

---

- **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



## How do entities communicate? A Protocol!








- 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

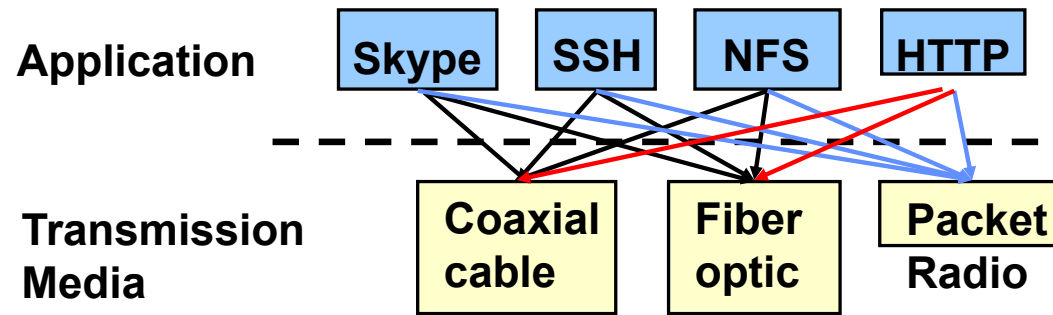
---

- 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.  Callee: "Hello?"
6. Caller: "Hi, it's Kubi...."  
Or: "Hi, it's me" (← what's *that* about?) 
7. Caller: "Hey, do you think ... blah blah blah ..." **pause**  

1. Callee: "Yeah, blah blah blah ..." **pause**
2. Caller: Bye 
3. Callee: Bye
4. Hang up 

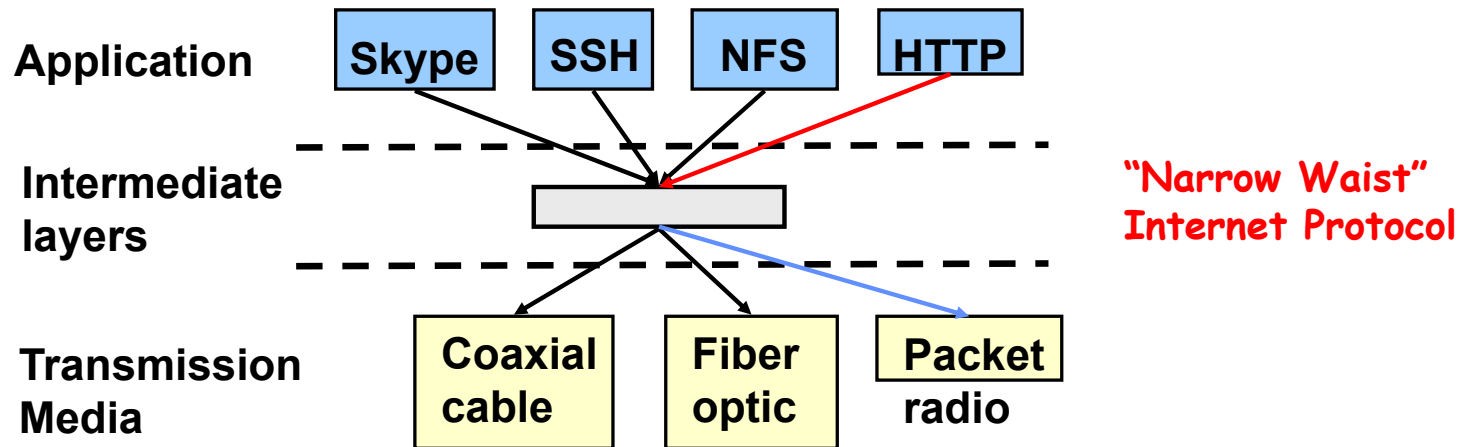
# Global Communication: The Problem

---



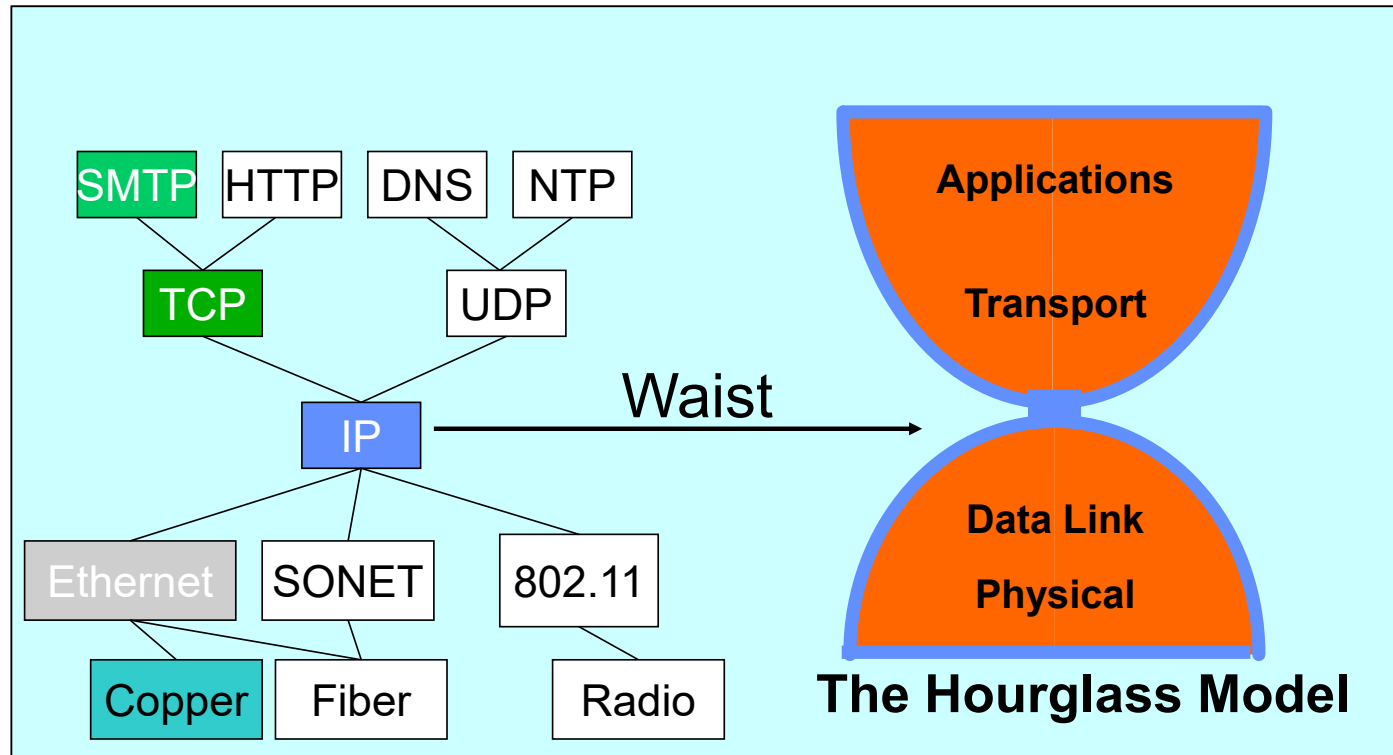
- Many different applications
  - email, web, P2P, etc.
- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?
  - Re-implement every application for every technology?
- No! But how does the Internet design avoid this?

## 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

# The Internet Hourglass



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

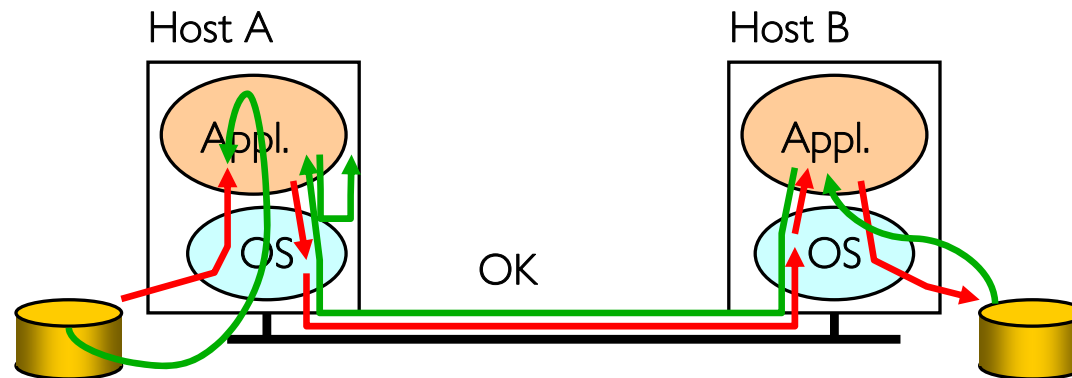
## End-To-End Argument

---

- Hugely influential paper: “End-to-End Arguments in System Design” by Saltzer, Reed, and Clark ('84)
  - Up on Resources page for those of you who are interest
- “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

## E2E 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???

## Administrivia

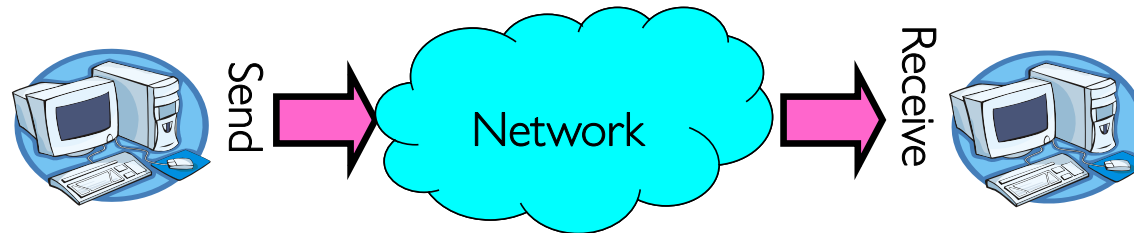
---

- Midterm 3: *Next Thursday!*
  - No class on Thursday. I'll have special office hours during class time.
  - Three double-sided pages of notes
  - Watch for Ed post about where you should go: we have multiple exam rooms
- All material up to Thursday's lecture is fair game
- Final deadlines during RRR week:
  - Yes, there will be office hours – watch for specifics
- Also – we have a special lecture during RRR week (just for fun) on Tuesday 5/5
  - During normal class time!

# 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**
    - » Over Internet, **destination** specified by **IP address** and **Port** (Recall Web server example!)
  - 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  $T1 \rightarrow T2$ 
  - $T1 \rightarrow \text{buffer} \rightarrow T2$
  - Very similar to producer/consumer
    - »  $\text{Send} = V, \text{Receive} = P$
    - » 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);  
}
```



Send  
Message

Consumer:

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



Receive  
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  $\equiv$  requester, Server  $\equiv$  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);
```

Request  
File

Get  
Response

Receive  
Request

Send  
Response

# 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

---

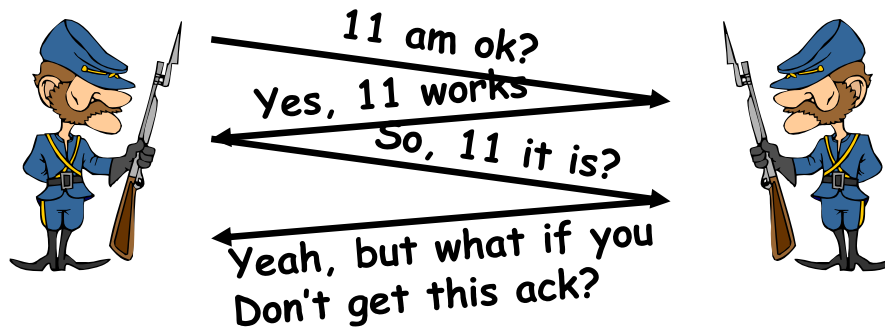
- General's paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, 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

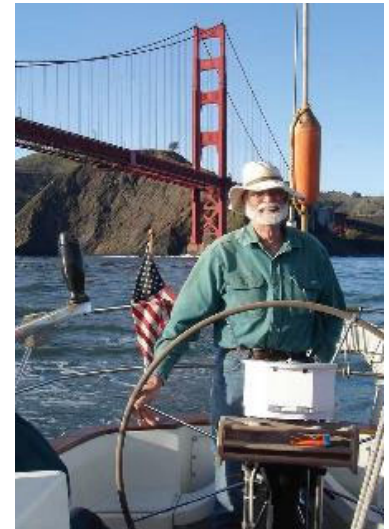


- No way to be sure last message gets through!
  - In real life, use radio for simultaneous (out of band) communication
- So, clearly, we need something other than simultaneity!

## 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



**Jim Gray**

## Two-Phase Commit Protocol: Summary

---

- 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:
  - Coordinator asks all participants either promise (vote) to **commit** or **abort** the **transaction**
  - Voters record promise in log, then acknowledge
- Commit Phase:
  - If **anyone** votes to abort, coordinator writes **"Abort"** in its log and tells everyone to abort; each records **"Abort"** in log
  - If **all** participants vote to **"Commit"**, 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

## Two-Phase Commit: Preparing

---

### If Worker Agrees to Commit

- Machine **guarantees** that it will accept transaction
- Must be **recorded in log** so machine will remember this decision if it fails and restarts

### If Worker Agrees to Abort

- Machine **guarantees** 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

---

### 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

### Commit Transaction

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

## Two-Phase Commit: Finishing

---

### Commit Transaction

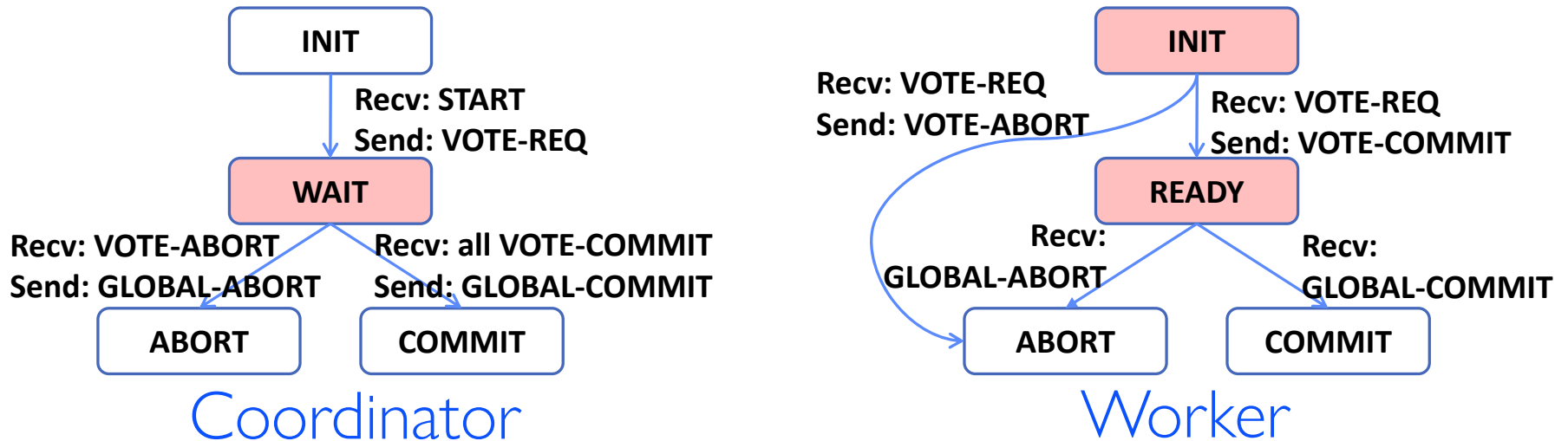
- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

### 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

## State Machine Description of 2PC



- Two Phase Commit (2PC) can be described with interacting state machines
- Coordinator only waits for votes in “WAIT” state
  - In WAIT, if doesn't receive  $N$  votes, it times out and sends GLOBAL-ABORT
- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-\* message in READY
  - Coordinator fails  $\Rightarrow$  workers BLOCK waiting for coordinator to recover and send GLOBAL\_\* message

## Detailed Algorithm

### 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

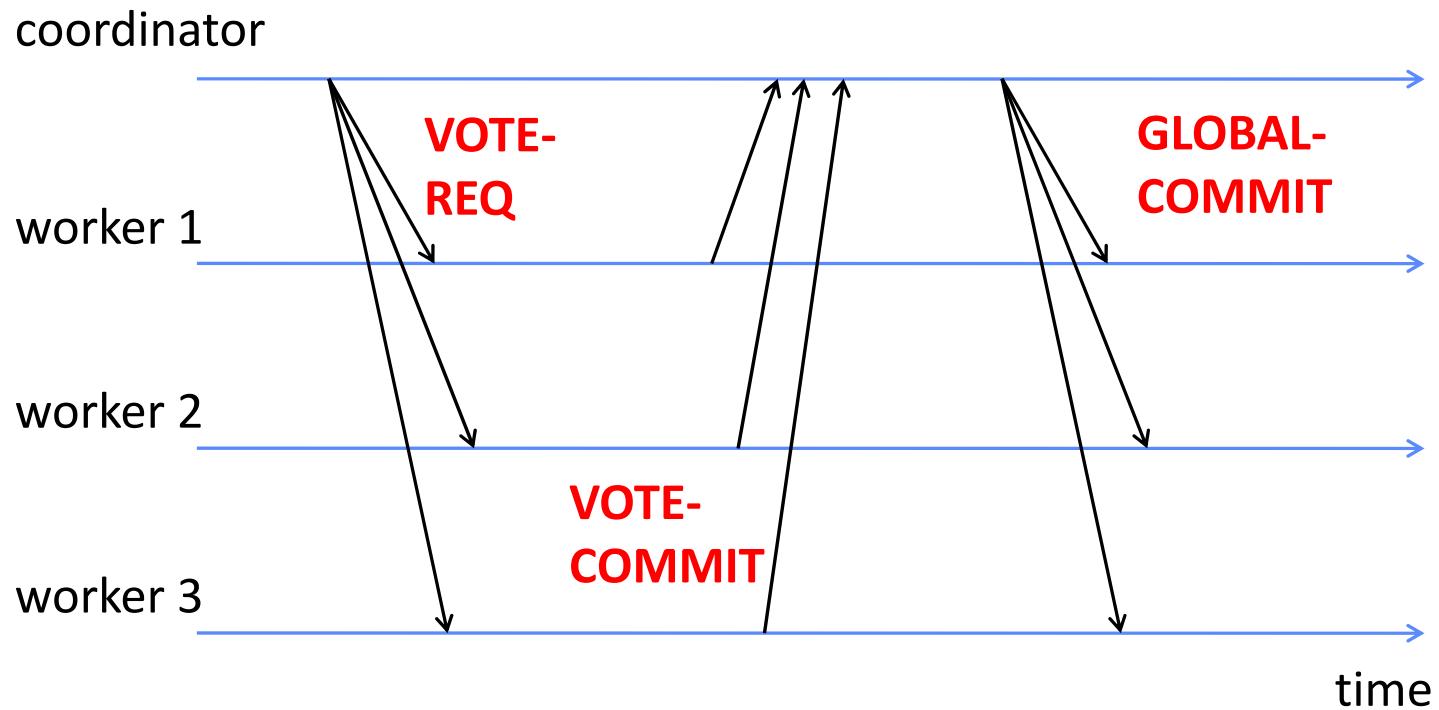
### Worker Algorithm

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

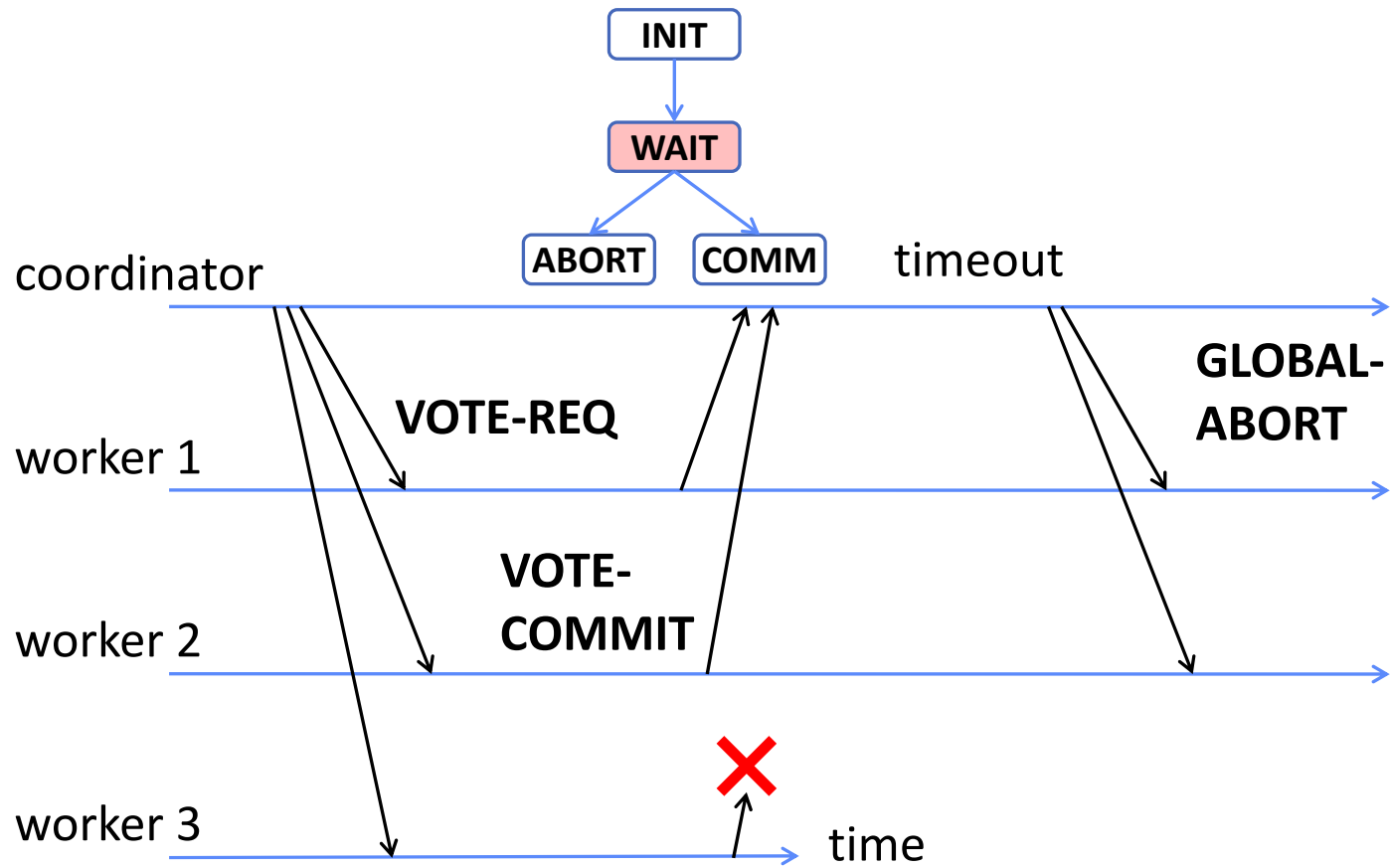
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

## Failure Free Example Execution

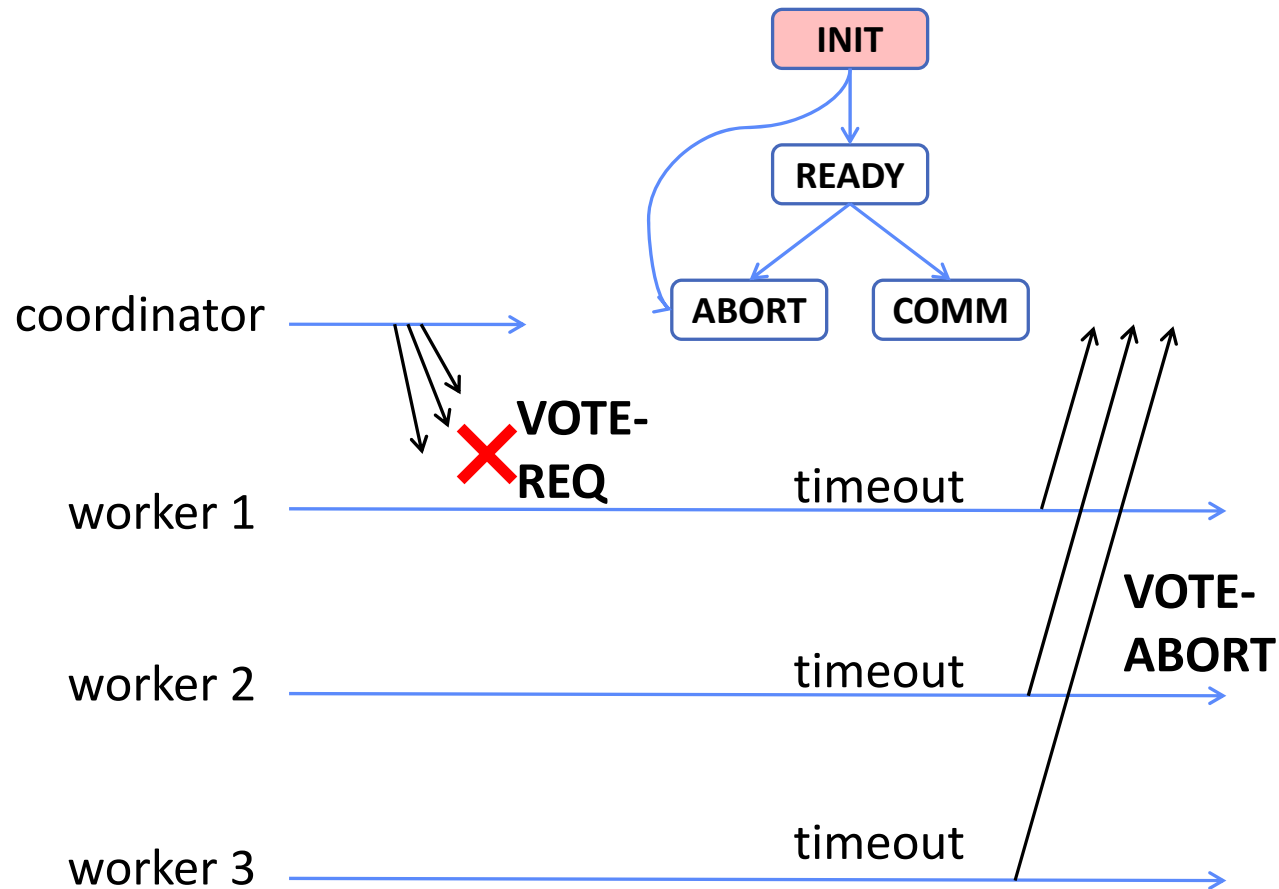
---



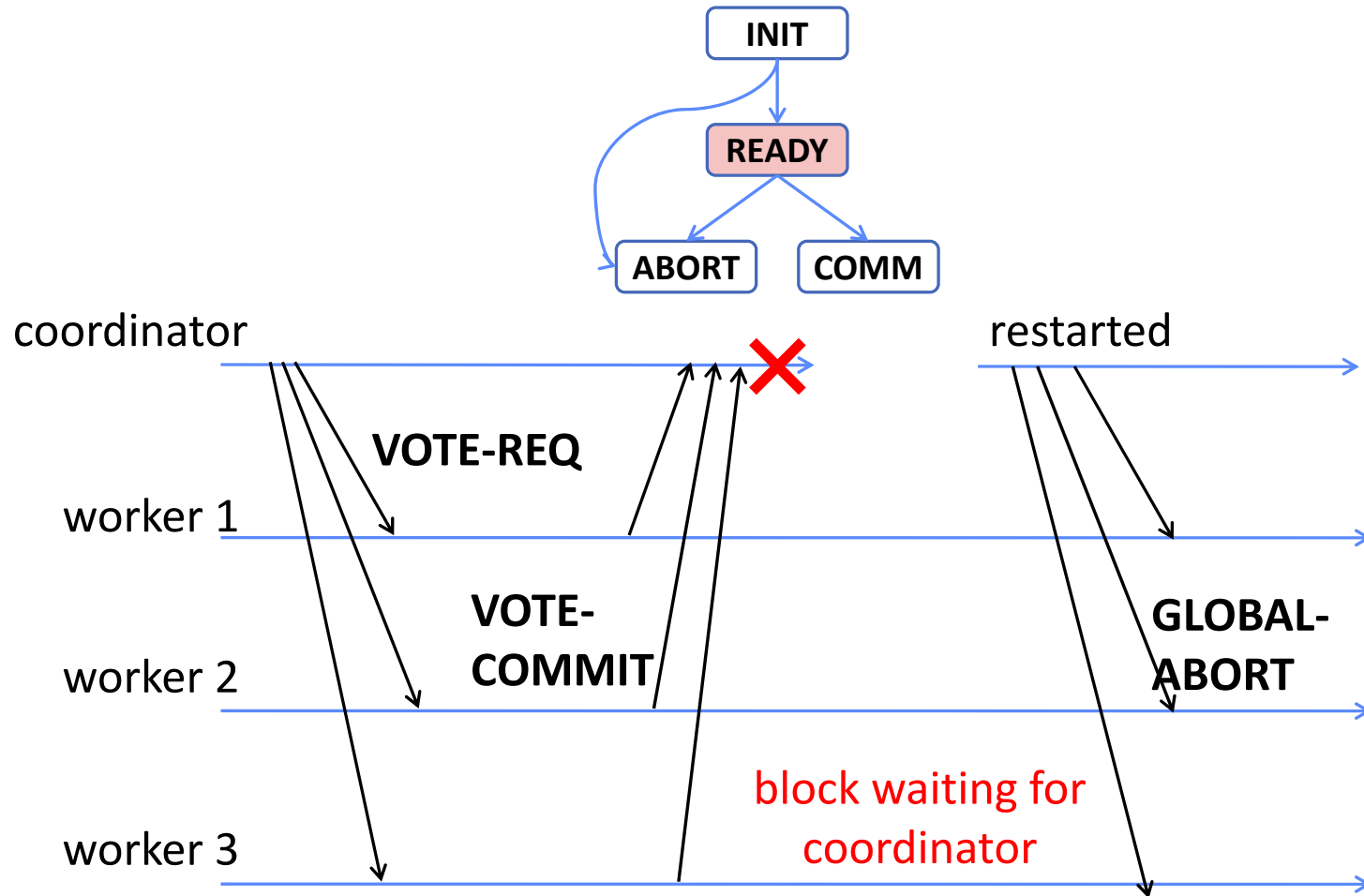
# Example of Worker Failure



## Example of Coordinator Failure #1



## Example of Coordinator Failure #2



## 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**

# Distributed Decision Making Discussion

---

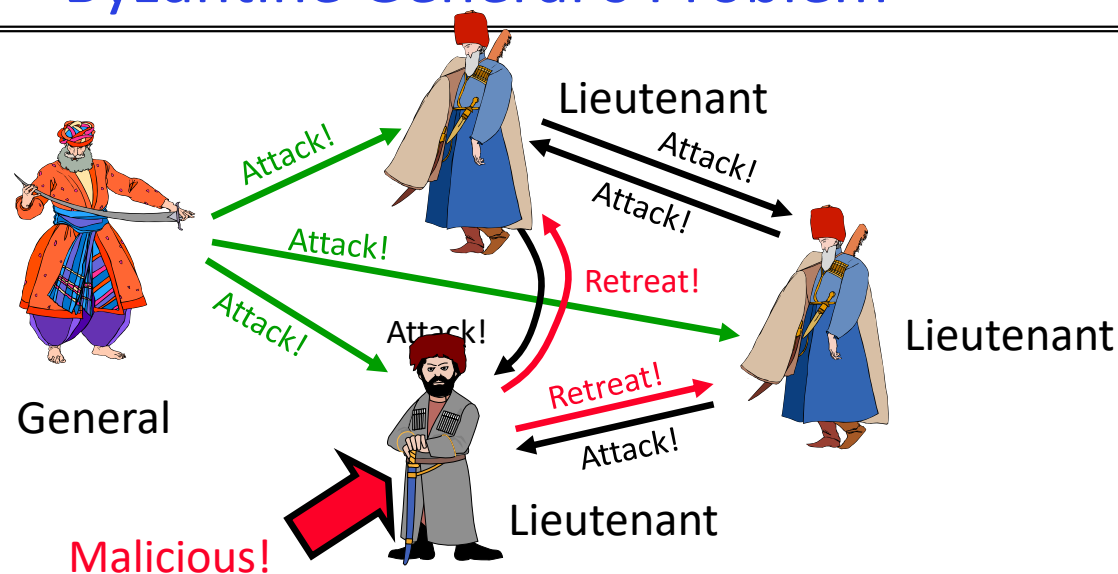
- Why is distributed decision making desirable?
  - Fault Tolerance!
  - A group of machines can come to a decision even if one or more of them fail during the process
    - » Simple failure mode called “failstop” (different modes later)
  - After decision made, result recorded in multiple places
- Undesirable 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

## Alternatives to 2PC

---

- **Three-Phase Commit:** One more phase, allows nodes to fail or block and still make progress.
- **PAXOS:** An alternative used by Google and others that does not have 2PC blocking problem
  - Develop by Leslie Lamport
  - No fixed leader, can choose new leader on fly, deal with failure
  - Some think this is extremely complex!
- **RAFT:** PAXOS alternative from John Ousterhout (Stanford)
  - Simpler to describe complete protocol
  - You are working on RAFT for HW6!
- What happens if one or more of the nodes is malicious?
  - **Malicious:** attempting to compromise the decision making
  - Use a more hardened decision making process:  
**Byzantine Agreement** and **Block Chains**

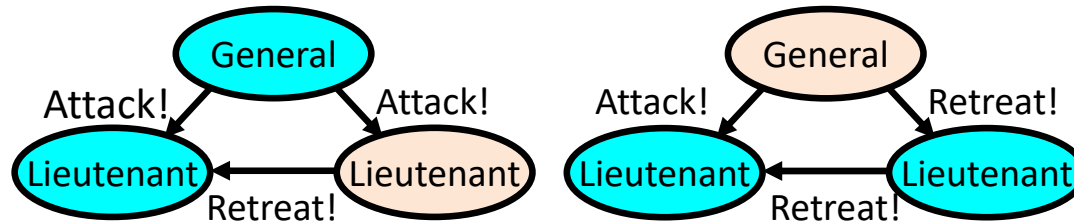
# Byzantine General's Problem



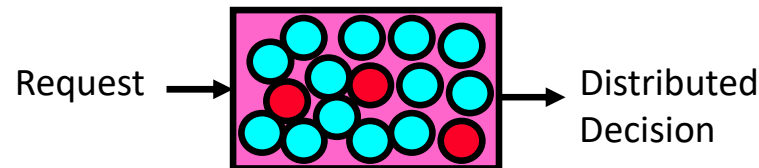
- Byzantine General's Problem ( $n$  players):
  - One General and  $n-1$  Lieutenants
  - Some number of these ( $f$ ) can be insane or malicious
- The commanding general must send an order to his  $n-1$  lieutenants such that the following Integrity Constraints apply:
  - IC1: All loyal lieutenants obey the same order
  - IC2: 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

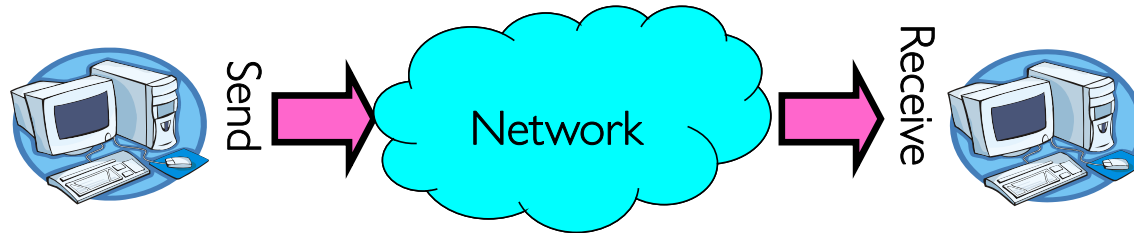


- With  $f$  faults, need  $n > 3f$  to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in  $n$
  - Newer algorithms have message complexity  $O(n^2)$ 
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them ( $< n/3$ ) are malicious



# How do we know that both sides speak same language?

---



- An object in memory has a machine-specific binary *representation*
  - Threads within a single process have the same view of what's in memory
  - Easy to compute offsets into fields, follow pointers, etc.
- In the absence of shared memory, externalizing an object requires us to turn it into a sequential sequence of bytes
  - **Serialization/Marshalling**: Express an object as a sequence of bytes
  - **Deserialization/Unmarshalling**: Reconstructing the original object from its marshalled form at destination

## Simple Data Types

---

`uint32_t x;`

- Suppose I want to write a `x` to a file
- First, open the file: `FILE* f = fopen("foo.txt", "w");`
- Then, I have two choices:
  1. `fprintf(f, "%lu", x);`
  2. `fwrite(&x, sizeof(uint32_t), 1, f);`
    - » Or equivalently, `write(fd, &x, sizeof(uint32_t));` (perhaps with a loop to be safe)
- Neither one is “wrong” but sender and receiver should be consistent!

# Machine Representation

---

- Consider using the machine representation:
  - `fwrite(&x, sizeof(uint32_t), 1, f);`
- How do we know if the recipient represents **x** in the same way?
  - For pipes, is this a problem?
  - What about for sockets?

# Endianness

- For a byte-address machine, which end of a machine-recognized object (e.g., int) does its byte-address refer to?
- **Big Endian**: address points to most-significant byte
- **Little Endian**: address points to least-significant byte

Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	Bi (Big/Little) Endian
ARM	Bi (Big/Little) Endian
IA-64 (64 bit)	Bi (Big/Little) Endian
MIPS	Bi (Big/Little) Endian

## Experiment:

```
int main(int argc, char *argv[])
{
    int val = 0x12345678;
    int i;
    printf("val = %x\n", val);
    for (i = 0; i < sizeof(val); i++) {
        printf("val[%d] = %x\n", i, ((uint8_t *) &val)[i]);
    }
}
```

## Result:

```
(base) CullerMac19:code09 culler$ ./endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
```

# What Endian is the Internet?

## NAME

arpa/inet.h - definitions for internet operations

## SYNOPSIS

```
#include <arpa/inet.h>
```

## DESCRIPTION

The `in_port_t` and `in_addr_t` types shall be defined as described in [<netinet/in.h>](#).

The `in_addr` structure shall be defined as described in [<netinet/in.h>](#).

The `INET_ADDRSTRLEN` <sup>[IP6]</sup> and `INET6_ADDRSTRLEN` macros shall be defined as described in [<netinet/in.h>](#).

The following shall either be declared as functions, defined as macros, or both. If functions are declared, function prototypes

```
uint32_t htonl(uint32_t);
uint16_t htons(uint16_t);
uint32_t ntohl(uint32_t);
uint16_t ntohs(uint16_t);
```

The `uint32_t` and `uint16_t` types shall be defined as described in [<inttypes.h>](#).

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
in_addr_t    inet_addr(const char *);
char         *inet_ntoa(struct in_addr);
const char   *inet_ntop(int, const void *restrict, char *restrict,
                        socklen_t);
int          inet_pton(int, const char *restrict, void *restrict);
```

Inclusion of the `<arpa/inet.h>` header may also make visible all symbols from [<netinet/in.h>](#) and [<inttypes.h>](#).

- **Big Endian**
  - Network byte order
  - Vs. “host byte order”

## Dealing with Endianness

---

- Decide on an “on-wire” endianness
- Convert from native endianness to “on-wire” endianness before sending out data (serialization/marshalling)
  - `uint32_t htonl(uint32_t)` and `uint16_t htons(uint16_t)` convert from native endianness to network endianness (big endian)
- Convert from “on-wire” endianness to native endianness when receiving data (deserialization/unmarshalling)
  - `uint32_t ntohl(uint32_t)` and `uint16_t ntohs(uint16_t)` convert from network endianness to native endianness (big endian)

## What About Richer Objects?

---

- Consider `word_count_t` of Homework 0 and 1 ...
- Each element contains:
  - An `int`
  - A *pointer* to a string (of some length)
  - A *pointer* to the next element
- `fprintf_words` writes these as a sequence of lines (character strings with `\n`) to a file stream
- What if you wanted to write the whole list as a binary object (and read it back as one)?
  - How do you represent the string?
  - Does it make any sense to write the pointer?

```
typedef struct word_count
{
    char *word;
    int count;
    struct word_count *next;
}
word_count_t;
```

# Data Serialization Formats

---

- JSON and XML are commonly used in web applications
- Lots of ad-hoc formats

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

```
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<glossary><title>example glossary</title>
<GlossDiv><title>S</title>
<GlossList>
  <GlossEntry ID="SGML" SortAs="SGML">
    <GlossTerm>Standard Generalized Markup Language</GlossTerm>
    <Acronym>SGML</Acronym>
    <Abbrev>ISO 8879:1986</Abbrev>
    <GlossDef>
      <para>A meta-markup language, used to create markup
languages such as DocBook.</para>
      <GlossSeeAlso OtherTerm="GML">
        <GlossSeeAlso OtherTerm="XML">
      </GlossDef>
      <GlossSee OtherTerm="markup">
    </GlossEntry>
  </GlossList>
</GlossDiv>
</glossary>
```

# Data Serialization Formats: Many Options

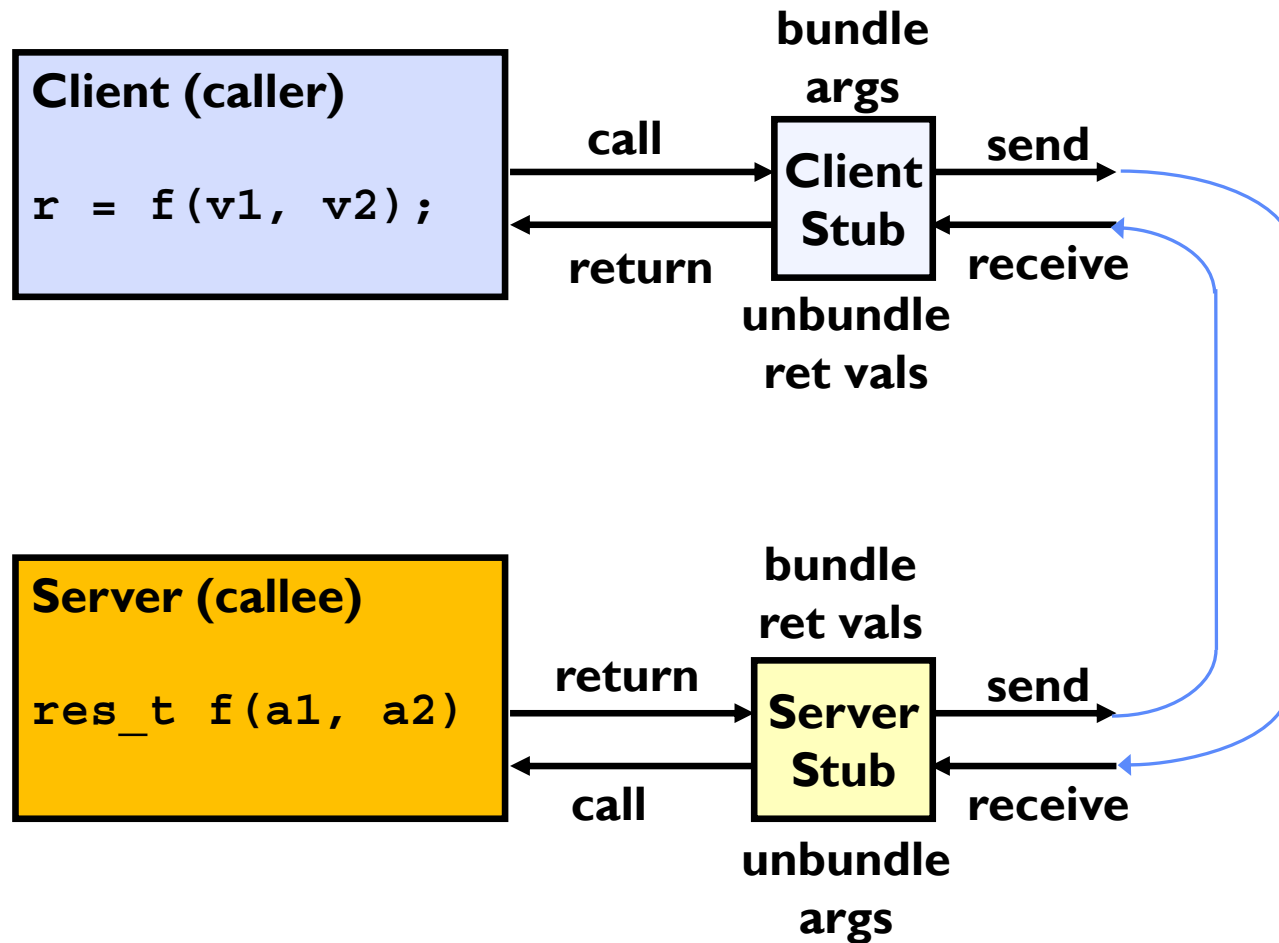
Name	Creator-maintainer	Based on	Standardized?	Specification	Binary?	Human-readable?	Supports references?	Schema-IDL?	Standard APIs	Supports Zero-copy operations
Apache Avro	Apache Software Foundation	N/A	No	Apache Avro™ 1.8.1 Specification	Yes	No	N/A	Yes (built-in)	N/A	N/A
Apache Parquet	Apache Software Foundation	N/A	No	Apache Parquet(1)	Yes	No	No	N/A	Java, Python	No
ASN.1	ISO, IEC, ITU-T	N/A	Yes	ISO/IEC 8824, X.680 series of ITU-T Recommendations	Yes (BER, DER, PER, CER, or custom via ECN)	Yes (XER, JER, GSER, or custom via ECN)	Partial	Yes (built-in)	N/A	Yes (OER)
Bencode	Bram Cohen (creator) BitTorrent, Inc. (maintainer)	N/A	De facto standard via BitTorrent Enhancement Proposal (BEP)	Part of BitTorrent protocol specifications	Partially (numbers and delimiters are ASCII)	No	No	No	No	N/A
Binn	Bernardo Ramos	N/A	No	Binn Specification	Yes	No	No	No	No	Yes
BSON	MongoDB	JSON	No	BSON Specifications	Yes	No	No	No	No	N/A
CBOR	Carsten Bormann, P. Hoffman	JSON (loosely)	Yes	RFC 7049	Yes	No	Yes through tagging	Yes ( CDDL )	No	Yes
Comma-separated values (CSV)	RFC author: Yakov Shafranovich	N/A	Partial (myriad informal variants used)	RFC 4180 (among others)	No	Yes	No	No	No	No
Common Data Representation (CDR)	Object Management Group	N/A	Yes	General Inter-ORB Protocol	Yes	No	Yes	Yes	ADA, C, C++, Java, Cobol, Lisp, Python, Ruby, Smalltalk	N/A
D-Bus Message Protocol	freedesktop.org	N/A	Yes	D-Bus Specifications	Yes	No	No	Partial (Signature strings)	Yes (see D-Bus)	N/A
Efficient XML Interchange (EXI)	W3C	XML, Efficient XML	Yes	Efficient XML Interchange (EXI) Format 1.0	Yes	Yes (XML)	Yes (XPointer, XPath)	Yes (XML Schema)	(DOM, SAX, StAX, XQuery, XPath)	N/A
FlatBuffers	Google	N/A	No	flatbuffers github page Specification	Yes	Yes (Apache Arrow)	Partial (internal to the buffer)	Yes (2)	C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C, Dart, Lua, TypeScript	Yes
Fast Infoset	ISO, IEC, ITU-T	XML	Yes	ITU-T X.891 and ISO/IEC 24824-1:2007	Yes	No	Yes (XPointer, XPath)	Yes (XML schema)	Yes (DOM, SAX, XQuery, XPath)	N/A
FHIR	Health_Level_7	REST basics	Yes	Fast Healthcare Interoperability Resources	Yes	Yes	Yes	Yes	Hapi for FHIR(1) JSON, XML, Turtle	No
Ion	Amazon	JSON	No	The Amazon Ion Specification	Yes	Yes	No	No	No	N/A
Java serialization	Oracle Corporation	N/A	Yes	Java Object Serialization	Yes	No	Yes	No	Yes	N/A
JSON	Douglas Crockford	JavaScript syntax	Yes	STD 96/RFC 8259 (ancillary: RFC 6901, RFC 6902), ECMA-404, ISO/IEC 21778:2017	No, but see BSON, Smile, UBJSON	Yes	Yes (JSON Pointer (RFC 6901); alternately: JSONPath, XPath, JSPON, jsonselect), JSON-LD	Partial (JSON Schema Proposal, ASN.1 with JER, Kwalify, Rxi, Itemsript Schema), JSON-LD	Partial (Clarinets, JSONQuery, JSONPath), JSON-LD	No
MessagePack	Sadyuki Furuhashi	JSON (loosely)	No	MessagePack format specification	Yes	No	No	No	No	Yes
Netstrings	Dan Bernstein	N/A	No	netstrings.txt	Yes	Yes	No	No	No	Yes
OGDL	Rolf Veen	?	No	Specification	Yes (Binary Specification)	Yes	Yes (Path Specifications)	Yes (Schema WD)		N/A
OPC-UA Binary	OPC Foundation	N/A	No	opcfoundation.org	Yes	No	Yes	No	No	N/A
OpenDDL	Eric Lengyel	C, PHP	No	OpenDDL.org	No	Yes	Yes	No	Yes (OpenDDL Library)	N/A
Pickle (Python)	Guido van Rossum	Python	De facto standard via Python Enhancement Proposals (PEPs)	(3) PEP 3154 - Pickle protocol version 4	Yes	No	No	No	Yes (4)	No
Property list	NeXT (creator) Apple (maintainer)	?	Partial	Public DTD for XML formats	Yes	Yes	No	?	Cocoa, CoreFoundation, OpenStep, GnuStep	No
Protocol Buffers (protobuf)	Google	N/A	No	Developer Guide: Encoding	Yes	Partial	No	Yes (built-in)	C++, C#, Java, Python, Javascript, Go	No

## Remote Procedure Call (RPC)

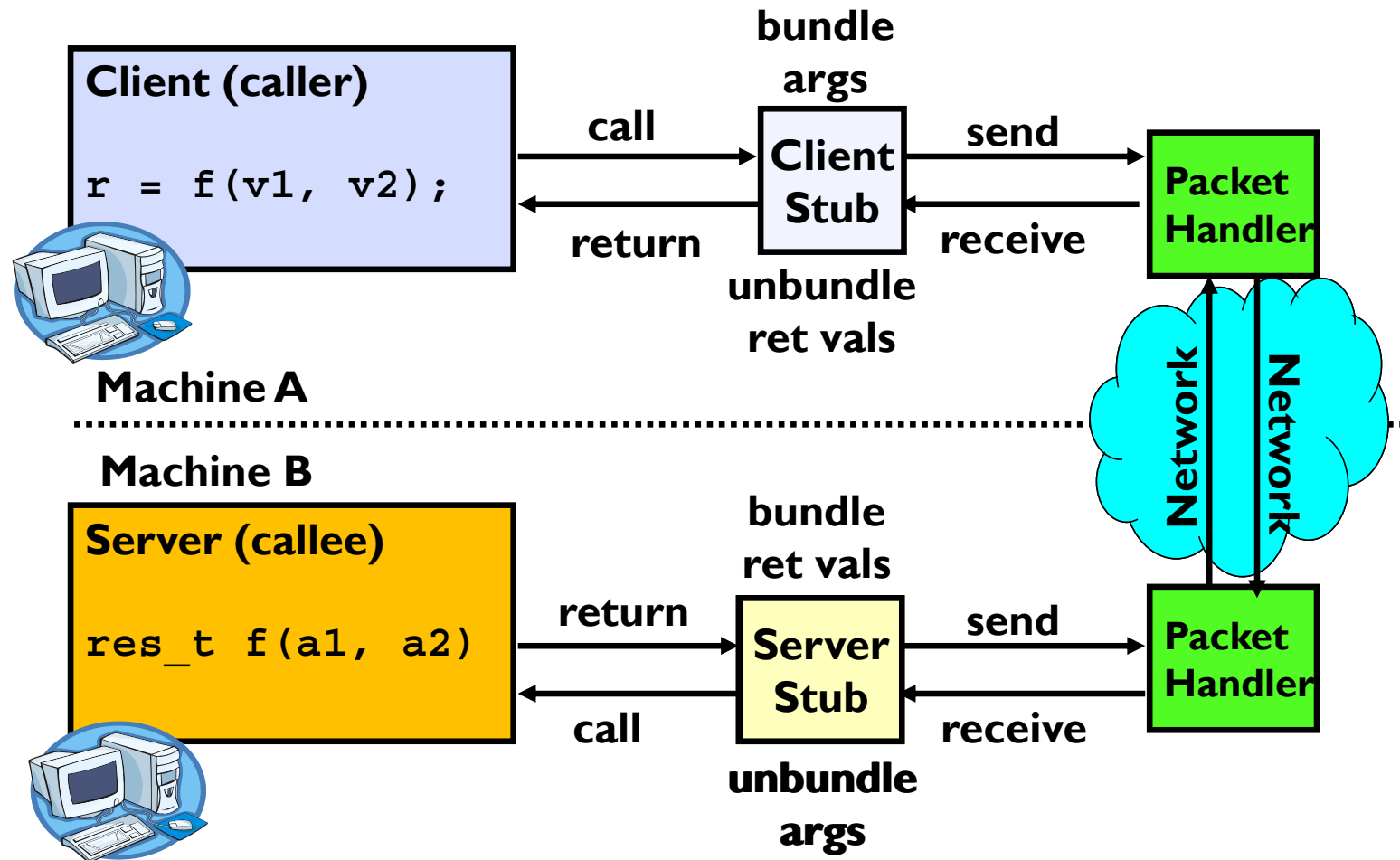
---

- 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
  - **And must deal with machine representation by hand**
- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Idea: Make communication look like an ordinary function call
  - Automate all of the complexity of translating between representations
  - Client calls:  
**remoteFileSystem→Read("rutabaga");**
  - Translated automatically into call on server:  
**fileSys→Read("rutabaga");**

# RPC Concept



# RPC Information Flow



## RPC 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.
  - Use of standardized **serialization** protocol

## RPC Details (1/3)

---

- Equivalence with regular procedure call
  - Parameters  $\Leftrightarrow$  Request Message
  - Result  $\Leftrightarrow$  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

## RPC Details (2/3)

---

- Cross-platform issues:
  - What if client/server machines are different architectures/ languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)
- How does client know which mbox (destination queue) 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

## RPC Details (3/3)

---

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service → 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 dist. 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

## Problems with RPC: Performance

---

- RPC is *not* performance transparent:
  - Cost of Procedure call  $\ll$  same-machine RPC  $\ll$  network RPC
  - Overheads: Marshalling, Stubs, Kernel-Crossing, Communication
- Programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex

## Cross-Domain Communication/Location Transparency

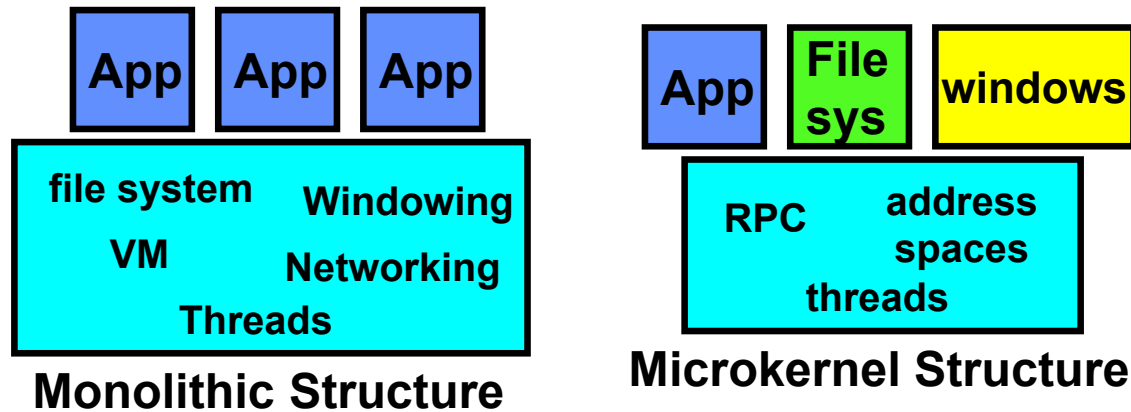
---

- 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 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



- 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.

## Conclusion (1/2)

---

- 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
- E2E argument encourages us to keep Internet communication simple
  - If higher layer can implement functionality correctly, implement it in a lower layer **only** if:
    - » it improves the performance significantly for application that need that functionality, and
    - » it **does not impose burden** on applications that do not require that functionality
- 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
- Two-phase commit: a form of distributed decision making
  - First, make sure everyone guarantees they will commit if asked (prepare)
  - Next, ask everyone to commit

## Conclusion (2/2)

---

- **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 or in remote domain
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
  - Adapts automatically to different hardware and software architectures at remote end
- **Next time: Distributed File System:**
  - Transparent access to files stored on a remote disk
  - Caching for performance