Bringing It Together

• Networks are collections of boxes (computers and specialized computers) and cables that move data
  – Networks themselves are disparate and have their own protocols
    • These protocols move data within a network
  – Internets smooth out those differences and allow us to communicate between the different networks
    • Internet protocols move data from source machine to destination machine
  – The Internet is an example of an internet
• When data arrives at a machine, the OS is responsible for delivering it to the correct process
  – UDP and TCP/IP help with this
  – TCP provides an illusion of reliability---the OS is responsible for implementing this reliability
  – With reliability comes danger: if we aren’t careful, we could overload an already congested network and cause it to collapse
  – TCP has algorithmic pieces to help with this: additive increase, multiplicative decrease; slow start; reaction to timeout events; round trip time variance (initial timer setting); and retransmit backoff
Transferring Data Over an Internet

LAN1

Host A

LAN1 adapter

protocol software

client

(1) data

internet packet

LAN1 frame

(2) data PH FH1

LAN2

Host B

LAN2 adapter

protocol software

server

(8) data

LAN2 frame

(7) data PH FH2

Router

LAN1 adapter

LAN2 adapter

protocol software

(4) data PH FH1

(5) data PH FH2

(6) data PH FH2

(3) data PH FH1

PH: Internet packet header
FH: LAN frame header
Network Costs

• Overhead: CPU time to put the data on or pull it off the wire
• Latency: time for one byte to go from one place to another
  – Latency from New York to San Francisco:
    • 3000 miles * 1 sec/186,282 miles = 15 ms
• Throughput: Maximum bytes per second (bandwidth)

• *Note: Bandwidth is not the whole story!*
• Key to good performance:
  – LAN: minimize overhead
  – WAN: keep the pipeline full!
Transferring Data Over an internet: Text Description

- Example has 2 LANs connected by a router
- Terminology:
  - PH: Internet packet header
  - FH: LAN frame header
- Steps to transfer:
  1. Data is sent from Host A’s client application to protocol software
  2. The PH and FH1 are appended to the data and sent along to the LAN1 adapter
  3. LAN1 adapter sends Data + PH + FH1 along to the router’s LAN1 adapter
  4. Data + PH + FH1 sent to router’s protocol software, which strips off FH1 and creates FH2
  5. Data + PH + FH2 sent to router’s LAN2 adapter
  6. Data + PH + FH2 sent to LAN2 adapter in LAN2
  7. Data + PH + FH2 sent along to protocol software where the PH and FH2 are stripped off
  8. Data goes along to Host B’s server application
Today’s Additions

• Accessing the Network
  – Sockets
    • Client-Side Programming
    • Server-Side Programming

• Parallel Computing
  – Description of Parallel Computing
  – Parallel Programming Models
    • Shared Memory
    • Message Passing

• Distributed Computing
  – Description of Distributed Computing
  – Unique Challenges in Distributed Computing
    • Event Ordering
    • Distributed Consensus
    • Atomicity of Transactions
Hardware and Software Organization of an Internet Application

![Diagram showing the hardware and software organization of an Internet Application.]

- **Internet client host**:
  - **Client**
  - **TCP/IP**
  - **Network adapter**
  - **Sockets interface (system calls)**
  - **Hardware interface (interrupts)**
  - **Global IP Internet**

- **Internet server host**:
  - **Server**
  - **TCP/IP**
  - **Network adapter**
  - **User code**
  - **Kernel code**
  - **Hardware and firmware**

- **Global IP Internet**
Internet Connections

- Clients and servers communicate by sending streams of bytes over connections
- A *socket* is an endpoint of a connection
  - Underlying basis for all Internet applications
  - Created in the early 80s as a part of the original Berkeley distribution of UNIX that contained an early version of the Internet protocols
  - Socket address is an IP address:port pair
- A *port* is a 16-bit integer that identifies a process:
  - *Ephemeral port*: Assigned automatically by client kernel when client makes a connection request.
  - *Well-known port*: Associated with some *service* provided by a server (e.g., port 80 is associated with Web servers)
Anatomy of an Internet Connection

Client socket address
128.2.194.242:51213

Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

Server socket address
208.216.181.15:80

Client host address
128.2.194.242

Server host address
208.216.181.15

51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers
Well-known Ports and Service Names

- Popular services have permanently assigned *well-known ports and corresponding well-known service names*:
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - Web servers: 80/http

- Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.
Sockets: Implementation Details

• What is a socket?
  – To the kernel, a socket is an endpoint of communication
  – To an application, a socket is a file descriptor that lets the application read/write from/to the network
    • Remember: All Unix I/O devices, including networks, are modeled as files

• Clients and servers communicate with each other by reading from and writing to socket descriptors

• The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors
1. Start server

- Server
- `open_listenfd`
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`

2. Start client

- Client
- `getaddrinfo`
- `socket`
- `connect`

3. Exchange data

- `send()`
- `recv()`

4. Disconnect client

- `close`

5. Drop client

- `recv()`
- `close`

Client / Server Session

Connection request

Await connection request from next client

EOF
Client / Server Session

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `rio_readlineb`
- `rio_writen`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `rio_readlineb`
- `rio_writen`
- `close`

**Sockets Interface**

- `open_clientfd`
- `open_listenfd`
- Await connection request from next client
- Connection request
Interfacing with DNS

Functions for retrieving host entries from DNS:

**getaddrinfo()**: query key is a DNS domain name, returns:

```c
struct addrinfo {
    int ai_flags;
    int ai_family;  /*desired address family*/
    int ai_socktype;  /*desired Layer 4 protocol*/
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```

**getnameinfo()**: query key is an IP address, returns a sockaddr struct (more soon)
Using `getaddrinfo()`

```c
int clientfd; /* socket descriptor */

/* setup information*/
struct addrinfo hints;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC; /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */

/* fill in the server's IP address and port */
struct addrinfo *result = NULL; /* DNS host entry */
int s = 0;
if ((s = getaddrinfo(hostname, port, &hints, &result)) != 0)
    return -1;
```

Note: This is untested code.
Client / Server Session

Client:
- `getaddrinfo`
- `socket`
- `connect`
- `send()`
- `recv()`
- `close`

Server:
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `recv()`
- `send()`
- `close`

`open_clientfd`}

`open_listenedfd`}

Connection request from next client

Await connection request from next client
**Client: `socket(2)`**

`socket(2)` creates a socket descriptor on the client

- Allocates and initializes some internal data structures
- `AF_UNSPEC`: indicates that the socket is associated with Internet protocols but either IPv4 or IPv6 is fine
- `SOCK_STREAM`: selects a reliable byte stream connection
  - Bi-directional pipes
  - Gives you TCP
- `SOCK_DGRAM` results in UDP

```c
int clientfd; /* socket descriptor */

if ((clientfd = socket(AF_UNSPEC, SOCK_STREAM, 0) <0)
    return -1; /* check_errno for cause of error */
...```
Socket Implementation in the OS

• Each socket fd has associated socket structure with:
  – Send and receive buffers
  – Queues of incoming connections (on listen socket)
  – A *protocol control block* (PCB)
  – A *protocol handle*

• PCB contains protocol-specific information, such as:
  – Pointer to IP Transmission Control Block with source/destination IP address and port
  – Information about received packets and position in stream
  – Information about unacknowledged sent packets
  – Information about timeouts
  – Information about connection state (setup/teardown)
Finally the client creates a connection with the server

– Client process blocks until the connection is created
– After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls (typically send/recv) on descriptor clientfd

```
int clientfd;    /* socket descriptor */
struct addrinfo *result; /* DNS host entry initialized in
                         * call to getaddrbyinfo()*/
typedef struct sockaddr SA; /* generic sockaddr */
...

/* Establish a connection with the server */
if (connect(clientfd, result->ai_addr, result->ai_addrlen) < 0)
    return -1;
```
Client / Server Session

### Client
- `getaddrinfo`
- `socket`
- `connect`
- `send()`
- `recv()`
- `close`

### Server
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `recv()`
- `send()`
- `close`

**Sockets Interface**
- `open_clientfd`
- `open_listenfd`
- **Await connection request from next client**

Diagram shows the interaction between client and server, with steps like socket creation, binding, listening, connecting, sending and receiving data, and closing connections.
**Server: bind(2)**

`bind()` associates the socket descriptor with the socket address (created similarly to that of the client)

```c
int listenfd;  /* listening socket */
struct addrinfo *result;  /* DNS host entry initialized in * call to getaddrbyinfo()*/
...
/* listenfd will be an endpoint for all requests to port on any IP address for this host */
if (bind(listenfd, rp->ai_addr, rp->ai_addrlen) < 0)
    return -1;
```
Server: `listen(2)`

- `listen()` indicates that this socket will accept connection (`connect`) requests from clients
  - Kernel assumes an active socket on the client end by default
- `LISTENQ` is a constant indicating how many pending requests are allowed

```c
int listenfd; /* listening socket */

/* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;
```

- We’re finally ready to enter the main server loop that accepts and processes client connection requests.
Client

- `open_clientfd`
- `getaddrinfo`
- `socket`
- `connect`
- `send()`
- `recv()`
- `close`

Server

- `open_listenfd`
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `recv()`
- `send()`
- `close`

Connection request

Await connection request from next client
Server: `accept (2)`

- `accept()` blocks waiting for a connection request

```c
int listenfd; /* listening descriptor */
int connfd;  /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
```

- `accept()` returns a **connected descriptor** (`connfd`) with the same properties as the **listening descriptor** (`listenfd`)
  - Returns when the connection between client and server is created and ready for I/O transfers
  - All I/O with the client will be done via the connected socket

- `accept()` also fills in client’s IP address
Connected vs. Listening Descriptors

• Listening descriptor
  – End point for client connection requests
  – Created once and exists for lifetime of the server

• Connected descriptor
  – End point of the connection between client and server
  – A new descriptor is created each time the server accepts a connection request from a client
  – Exists only as long as it takes to service client

• Why the distinction?
  – Allows for concurrent servers that can communicate over many client connections simultaneously
    • E.g., Each time we receive a new request, we create a new thread to handle the request
accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`
The listen() call is listening on a different port than the port the server will eventually use to send data to the client.

A. True

B. False
1. Start server
   - Server
   - open_listenfd
   - getaddrinfo
   - socket
   - bind
   - listen
   - accept
   - recv
   - send

2. Start client
   - Client
   - open_clientfd
   - getaddrinfo
   - socket
   - connect
   - recv
   - send
   - close

3. Exchange data
   - Await connection request from next client
   - Connection request

4. Disconnect client

5. Drop client
Parallel Programming
Why a different type of programming?

- Sequential programs get no benefit from multiple processors.
- Many applications can be (re)designed/coded/compiled to generate cooperating, parallel instruction streams.
- No automated compiler/language exists to automate this “parallelization” process.
- Key property is how much communication per unit of computation.
  - The less communication per unit computation the better the scaling properties of the algorithm.
Parallel Programming

• Parallel programming involves:
  – Decomposing an algorithm into parts
  – Distributing the parts as tasks which are worked on by multiple processes simultaneously
  – Coordinating work and communication of those processes
    • Synchronization

• Parallel programming considerations:
  – Type of parallel architecture being used
  – Type of process communication used
Parallel Programming Models

Communication and synchronization based on either:

– Shared memory
  • Interprocess communication is implicit
  • Synchronization is explicit
  • Assume processes/threads can read & write a set of shared memory locations
  • Difficult to provide across machine boundaries

– Message passing
  • Interprocess communication is explicit
  • Synchronization is implicit
  • Extensible to communication in distributed systems
Shared Memory Programming Model

- Programs/threads communicate/cooperate via loads/stores to memory locations they share.
- Communication is therefore at memory access speed (very fast) and is implicit.
- Cooperating pieces must all execute on the same system (computer).
- OS services and/or libraries used for creating tasks (processes/threads) and coordination (semaphores/barriers/locks.)
Example Shared Memory Code

fork N processes
each process has a number, p, and computes
    istart[p], iend[p], jstart[p], jend[p]
for(s=0;s<STEPS;s++) {
    k = s&1; m = k^1;
    forall(i=istart[p];i<=iend[p];i++) {
        forall(j=jstart[p];j<=jend[p];j++) {
            a[k][i][j] = c1*a[m][i][j] + c2*a[m][i-1][j] +
                        c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                        c5*a[m][i][j+1]; // implicit comm
        }
    }
    barrier();
}
Message Passing Programming Model

• “Shared” data is communicated using send/receive services (across an external network).
• Shared data must be formatted into message chunks for distribution
  – Unlike shared memory
• Coordination is also via sending/receiving messages.
• Program components can be run on the same or different systems, so can use many of processes.
• Standard libraries exist to encapsulate messages:
  – Parasoft's Express (commercial)
  – PVM (standing for Parallel Virtual Machine, non-commercial)
  – MPI (Message Passing Interface, also non-commercial).
Message Passing Logistics: Synchronization

When do send/receive operations terminate?

Blocking (aka Synchronous):
Sender waits until its message is received
Receiver waits if no message is available

Non-blocking (aka Asynchronous):
Send operation “immediately” returns
Receive operation returns if message is available or not (polling)

Partially blocking/non-blocking:
send()/receive() with timeout
Limitations of Message Passing

Easy for OS, hard for programmer

– Programmer must code communication
– Programmer may have to code format conversions, flow control, error control
– No dynamic resource discovery
Parallel Computing
Parallel Computing: Description

• Tightly-coupled systems

• Processors share:
  – Clock
  – Memory (a single physical address space)
  – A single OS

• Examples: Multicore systems, Symmetric Multi Processor (SMP) systems (and you can make SMP multicore processors... )
Single CPU

Processor

Cache memories

Register file

ALU

Bus interface

Bus

Main memory
Symmetric Multi Processor

Processor 0

- Registers
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache

ALU

…

Processor 3

- Registers
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache

ALU

Main memory

Bus
Multicore Processor

Processor package

Core 0

Registers

L1 d-cache

L1 i-cache

L2 unified cache

L3 unified cache
(shared by all cores)

Core 3

Registers

L1 d-cache

L1 i-cache

L2 unified cache

Main memory
Distributed Computing
What is a Distributed System?

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

Leslie Lamport
Distributed Computing: Description

- Loosely-coupled systems
- A set of physically separate processors connected by one or more communication links
  - Nodes connected by a network
- As a result:
  - Each processor has its own memory
  - Each processor runs an independent OS
  - Each processor has its own clock
- Communication is more expensive than in parallel computing
- Examples:
  - Email, File servers, Printer access, Backup over network, WWW
  - Supercomputers, Clusters, Massively Parallel Machines
A Cluster

64 dual-processor PCs connected by a 100Mb/Sec Ethernet switch
Stampede: A TACC Supercomputer
Very Distributed Computing

• Grid computing
  – Multiple locations
  – Heterogeneous architectures
  – Example: XSEDE

• Cloud computing
Unique Challenges Provided by Distributed Computing
Event Ordering

• Coordination of requests (especially in a fair way) requires events (requests) to be ordered.

• Stand-alone systems/Parallel Computing:
  – Shared Clock / Memory
  – Use a time-stamp to determine ordering

• Distributed Systems: What does time mean?
  – No global clock
  – Each clock runs at different speeds (clock drift)

• How do we order events executing on physically separated systems?
  – Recall that communication is through message passing... does that help us?
Event Ordering: Distributed Systems

• A message must be sent before it can be received
• Send/Receives can thus “synchronize” the clocks
Event Ordering: Happened-Before Relation

1. If A and B are events in the same process, and A executed before B, then $A \rightarrow B$.

2. If A is a message send and B is when the message is received, then $A \rightarrow B$.

3. $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
Happened-Before Relation

Ordered events
- $p_1$ precedes ___
- $q_4$ precedes ___
- $q_2$ precedes ___
- $p_0$ precedes ___

Unordered (Concurrent) events
- $q_0$ is concurrent with ___
- $q_2$ is concurrent with ___
- $q_4$ is concurrent with ___
- $q_5$ is concurrent with ___
We are mapping the events of 3 different processes, P, Q, and R, over time.
For each process the events happen in numerical order:
  - E.g., p0 happens before p1 which happens before p2, etc.
We also map the messages sent by processes at certain events and received by others in order to determine the relative order of different events.
Events by process:
  - Process P: p0, p1, p2, p3 and p4
  - Process Q: q0, q1, q2, q3, q4 and q5
  - Process R: r0, r1, r2 and r3
Messages between processes’ events:
  - p1 sent a message to q3 (as in Process P sends a message to Process Q at event p1 and it is received at Q’s event q3)
  - q2 sent a message to r2
  - q4 sent a message to r3
  - r0 sent a message to q1
  - r1 sent a message to q5
Ordered events (fill in the blank)
  - p1 precedes
  - q4 precedes
  - q2 precedes
  - p0 precedes
Unordered (concurrent) events (fill in the blank)
  - q0 is concurrent with
  - q2 is concurrent with
  - q4 is concurrent with
  - q5 is concurrent with
Happened-Before and Total Event Ordering

Define a notion of event ordering such that:

1. If $A \rightarrow B$, then $A$ precedes $B$.
2. If $A$ and $B$ are concurrent events, then nothing can be said about the ordering of $A$ and $B$.

Solution:

1. Each process $i$ maintains a *logical clock*, $LC_i$
2. When an event occurs locally, $LC_i = LC_i + 1$
3. When process $X$ sends a message to $Y$, it also sends a *time stamp* $LC_x$ in the message.
4. When $Y$ receives this message, it:
   
   $LC_y = LC_y + 1$
   
   if $LC_y < (LC_x + 1)$ //if $Y$’s clock is behind $X$’s
   
   $LC_y = LC_x + 1; //set Y’s to X’s value + 1$
Logical Clocks: Example

[Diagram showing logical clock example with processes and time progression.]

[Lamport 78]
Logical Clocks: Example

Plain Text

- We have 3 different processes 0, 1, and 2, and we are determining their logical clock values at each send/receive
  - In this scenario, we are only concerned with the send and receive events—there are no other local events
- Different clock values indicate that the events must happen in that order based on their happened-before relationships
- Order of events in the system:
  1. Process 0 sends a message to Process 1
     - Send is Event 1 on Process 0
     - Receive is Event 2 on Process 1
     - Process 1 does not have an Event 1. Its logical clock’s initial value is set according to Process 0’s and the rules of logical clocks.
  2. Process 0 sends a message to Process 2
     - Send is Event 2 on Process 0
     - Receive is Event 3 on Process 2
     - Process 2 does not have Events 1 or 2. Its logical clock’s initial value is set according to Process 0’s and the rules of logical clocks.
  3. Process 1 sends a message to Process 0
     - Send is Event 3 on Process 1
     - Receive is Event 4 on Process 0
     - Process 0 does not have an Event 3. Its logical clock is updated according to the value of Process 1’s clock and the rules of logical clocks.
  3. Process 2 sends a message to Process 1
     - Send is Event 4 on Process 2
     - Receive is Event 5 on Process 1
     - Process 1 does not have an Event 4. Its logical clock is updated according to the value of Process 2’s clock and the rules of logical clocks.
  4. Process 1 sends a message to Process 2
     - Send is Event 6 on Process 1
     - Receive is Event 7 on Process 2
     - Process 2 does not have Events 5 or 6. Its logical clock is updated according to the value of Process 1’s clock and the rules of logical clocks.
  5. Process 1 sends a message to Process 0
     - Send is Event 7 on Process 1
     - Receive is Event 8 on Process 0
     - Process 0 does not have Events 5, 6, or 7. Its logical clock is updated according to the value of Process 1’s clock and the rules of logical clocks.
If $A \rightarrow B$ and $C \rightarrow B$, can we say $A \rightarrow C$?

A. Yes
B. No
Atomicity
Atomicity in Distributed Systems

• How can we atomically update state on two different systems?
• Examples:
  – Atomically move a file from server A to server B
  – Atomically move $100 from one bank to another
• Issues:
  – Messages exchanged by systems can be lost
  – Systems can crash
• Can we use messages and retries over an unreliable network to synchronize the actions of two machines?
Distributed Consensus: The Generals’ Paradox

Problem:

- Two generals are on two separate mountains
- Can communicate only via messengers; but messengers can get lost or captured by enemy
- Goal is to coordinate their attack
  - If attack at different times → they lose!
  - If attack at the same time → they win!

Even if all previous messages get through, the generals still can’t coordinate their actions, since the last message could be lost, always requiring another confirmation message.
The Generals’ Paradox: Plain Text

Problem:
- 2 generals are on 2 separate mountains
- Can communicate only via messages; but messengers can get lost or captured by enemy
- Goal is to coordinate their attack
  - If attack at different times they both lose!
  - If attack at the same time they win!
- BUT how can they be really sure that the messages are being received?
  - Confirmation messages?
    - How do you know the last confirmation message was received? Results in infinite pinging back and forth
    - Even if all previous message get through, the generals still can’t coordinate their actions, since the last message could be lost, always requiring another confirmation message
iClicker Question

Distributed consensus in the presence of link failures is:
A. Possible
B. Not possible
Distributed Consensus with Link Failures

• Problem:
  – Take any exchange of messages that solves the general’s coordination problem.
  – Take the last message $m_n$. Since $m_n$ might be lost, but the algorithm still succeeds, it must not be necessary.
  – Repeat until no messages are exchanged.
  – No messages exchanged can’t be a solution, so our assumption that we have an algorithm to solve the problem must be wrong.

• Distributed consensus in the presence of link failures is impossible.
  – That is why timeouts are so popular in distributed algorithms.
  – Success can be probable, just not guaranteed in bounded time.
Distributed Transactions

• Two machines agree to do something, or not do it, atomically
  – they are not necessarily agreeing to do it at the same time

• The *two-phase commit protocol* allows coordination under reasonable operating conditions
  – Uses log on each machine to track whether commit happened
Two-Phase Commit Protocol: Phase 1

Phase 1: Coordinator requests a transaction
- Coordinator sends a REQUEST to all participants for that transaction
  
  Example: $C \rightarrow S1 : \text{“delete foo from /”}$
  $C \rightarrow S2 : \text{“add foo to /quux”}$
- On receiving request, participants perform these actions:
  - “Execute” the transaction locally
    - this may mean just writing it to a log
  - Write VOTE_COMMIT or VOTE_ABORT to their local logs
  - Send VOTE_COMMIT or VOTE_ABORT to coordinator
Two-Phase Commit Protocol: Phase 2

Phase 2: Coordinator commits or aborts the transaction

• Coordinator decides
  – Case 1: Coordinator receives VOTE_ABORT or times-out
    • Coordinator writes GLOBAL_ABORT to log and sends GLOBAL_ABORT to participants
  – Case 2: Coordinator receives VOTE_COMMIT from all participants
    • Coordinator writes GLOBAL_COMMIT to log and sends GLOBAL_COMMIT to participants

• Participants commit the transaction
  – On receiving a decision, participants write GLOBAL_COMMIT or GLOBAL_ABORT to log
Does Two-Phase Commit Work?

• Yes ... can be proved formally

• Consider the following cases:
  
  – What if participant crashes during the request phase before writing anything to log?
    • On recovery, participant does nothing; coordinator will timeout and abort transaction; and retry!

  – What if coordinator crashes during phase 2?
    On restart:
    • Case 1: Log does not contain GLOBAL_* → send GLOBAL_ABORT to participants and retry
    • Case 2: Log contains GLOBAL_ABORT → send GLOBAL_ABORT to participants
    • Case 3: Log contains GLOBAL_COMMIT → send GLOBAL_COMMIT to participants
iClicker Question

Can the Two-Phase Commit protocol fail to terminate?

A. Yes
B. No
Limitations of Two-Phase Commit

• What if the coordinator crashes during Phase 2 (before sending the decision) and does not wake up?
  – All participants block forever!
    (They may hold resources – e.g. locks!)

• Possible solution:
  – Participant, on timing out, can make progress by asking other participants (if it knows their identity)
    • If any participant had heard GLOBAL_ABORT → abort
    • If any participant sent VOTE_ABORT → abort
    • If all participants sent VOTE_COMMIT but no one has heard GLOBAL_* → can we commit?
      – NO – the coordinator could have written GLOBAL_ABORT to its log (e.g., due to local error or a timeout)
Two-Phase Commit: Summary

• Message complexity $3(N-1)$
  – Request/Reply/Broadcast, from coordinator to all other nodes.
• When you need to coordinate a transaction across multiple machines ... Use two-phase commit
  – For two-phase commit, identify circumstances where indefinite blocking can occur
  – Decide if the risk is acceptable
• If two-phase commit is not adequate, then ...  
  – Use *advanced distributed coordination techniques*
  – To learn more about such protocols, take a distributed computing course
Summary

• Accessing the network from user code is detailed but often formulaic

• Parallel Computing: tightly-coupled systems

• Parallel Programming Models
  – Shared Memory (OpenMP)
    • Sharing is implicit, synchronization is explicit
  – Message Passing (MPI)
    • Sharing is explicit, synchronization is implicit
Summary

• Distributed Computing: loosely-coupled systems

• Distributed systems may need to:
  – have a sense of time: use the happened before relationship to order events
  – provide atomicity: use two-phase commit protocol (at some level, the network is unreliable!)
  – elect a leader: use the bully algorithm
Announcements

• Project 4 (Last one!) due Friday, 5/10
  – Code at 5:59p, Design at 11:59p
  – No slip days!
  – Group registration due TODAY

• If you have a conflict for the final, you should have already contacted me (email, please!)
  – Thursday, 5/16, 2p-5p in FAC 21

• Next time: distributed computing concepts applied to file systems!