Networks and distributed computing

- Abstractions provided for networks

network card has fixed MAC address ->

deliver message to computer on LAN ->

machine-to-machine communication ->

unordered messages ->

unreliable messages ->

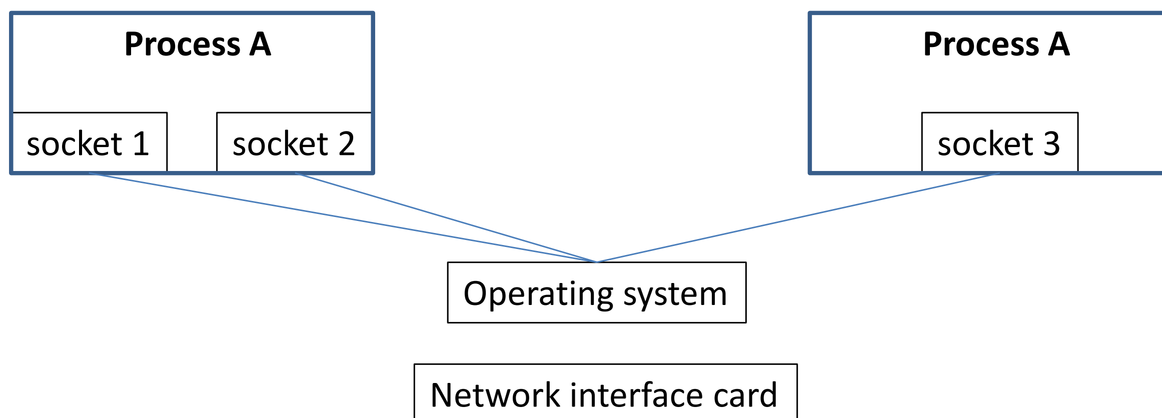
finite-sized messages ->

multiple computers ->

Machine-to-machine communication (IP) -> Process-to-process communication (sockets)

- Process abstraction: each process thinks it has its own:
  - Multiprocessor (threads)
  - Memory (address space)
  - Network interface cards (sockets)
- Socket
  - Virtual network interface card
  - Endpoint for communication
  - NIC named by MAC address; socket named by “port number” (via bind)
  - Programming interface: BSD sockets

Operating system multiplexes multiple sockets onto a single network interface card



- UDP (user datagram protocol): IP + sockets
- TCP (transmission control protocol): IP + sockets + reliable, ordered streams

Ordered messages

- Hardware interface: network can re-order messages sent by IP layer
  - Send A, B
  - Receive B, A
- Application interface: all messages are received in the order in which they were sent
- How to provide ordered messages?

- To have a notion of order, we must first have the notion of a network "connection" (so we know that a set of messages are related)
  - TCP: process opens connection (via `connect`), sends sequence of messages, then closes connection.
  - Sequence number of specific to a socket-to-socket connection

Reliable messages

- Hardware interface: networks can drop, duplicate, or corrupt messages
- Application interface: each message is delivered exactly once (without corruption)
- How to fix a dropped message?

- How does sender know message was dropped?

Reliable messages

- Duplicate messages
  - Detect by examining sequence #s
  - Fix by dropping the duplicate message

- Corrupted messages
  - Detect by adding redundant information (e.g., checksum)
  - Fix by dropping corrupted message

- Transform corrupted messages into dropped messages
- Transform potential dropped messages into potential duplicates
- Solve duplicates by dropping messages

Byte streams

- Hardware interface: send information over network in distinct messages
- Application interface: send data in a continuous stream (similar to reading/writing a file)
- TCP provides byte streams instead of distinct messages
- Sender sends messages of arbitrary size, which are combined into a single stream
- TCP implementation
  - Break up stream into fragments
  - Sends fragments as distinct messages
  - Reassembles fragments at destination

Message boundaries

- TCP has no message boundaries (in contrast, UDP preserves message boundaries)
  - E.g., sender sends 100 bytes, then 50 bytes. Receiver could receive 1-150 bytes.
- If receiver wants to receive a specific number of bytes, it can loop around receive call, or specify MSG_WAITALL
- How to know # of bytes to receive?

Client-server

- Different ways to structure a distributed application. Most common is client-server.
  - Server provides some centralized service
  - Client makes request to server, then waits for response
- E.g., web server
  - Server stores and generates web pages
  - Clients run web browsers
  - Requests are GET or POST
- E.g., producer-consumer with client-server
  - Server manages state associated with coke machine
  - Clients call client_produce() or client_consume(). These send request to the server and return when done.
  - Client calls may block at the server if request cannot be satisfied immediately.

Producer-consumer as a client-server distributed application

```
client_produce()
{
        send produce request to server
        wait for response
}

server()
{
        receive request
        if (produce request) {
                add coke to machine
        } else {
                take coke out of machine
        }
}
```

- Problems?

- How to fix?

Producer-consumer as a client-server distributed application

```
server()
{
        receive request
        if (produce request) {
                create thread that calls server_produce()
        } else {
                create thread that calls server_consume()
        }
}

server_produce()
{
        lock
        while (machine is full) {
                wait
        }
        put coke in machine
        send response to client
        unlock
}
```


Producer-consumer as a client-server distributed application

- This creates a new thread for each request. How to lower the overhead of creating threads?

- Other ways to structure the server
  - Polling (via `select`)
  - Signals

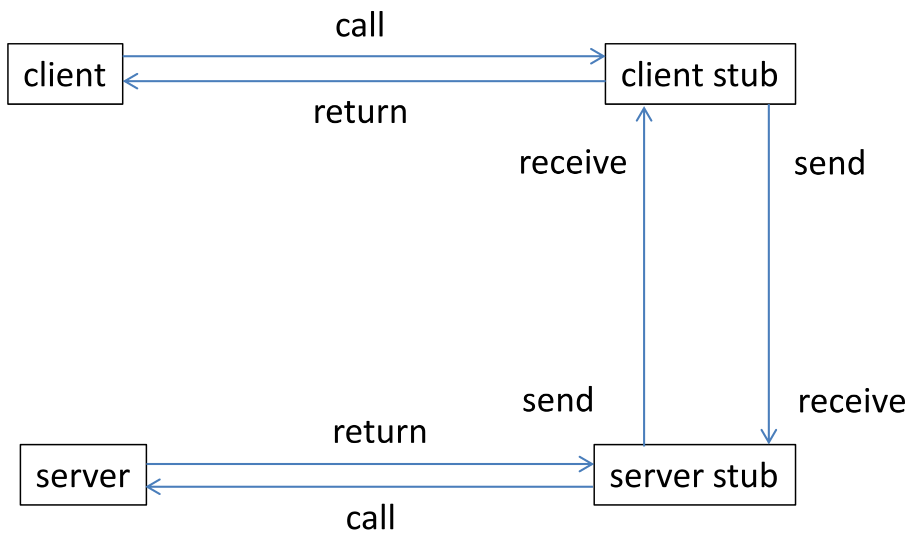
RPC

- We’ve been using send/receive to communicate between client and server
  - Client sends request to server
  - Server receives and processes request, then sends response to client
  - Client receives response from server
  - This exposes the distributed nature of the system to the programmer
  - Would like to make building a distributed application more similar to building a local application.
- What else in programming is similar to making a request to a server and getting a response?

RPC

- How to make message send/receive look like function call/return to both the client and the server?
  - Client wants the sending of its request to look like calling a function, and wants the reply from the server to look like returning from the function call.
  - Server wants the receiving of a request from the client to look receiving a function call, and wants the send of the response to look like returning to the caller.

Providing RPC abstraction via stub functions on client and server



RPC stubs

- Client stub

- Server stub

Producer-consumer using RPC

- Use reliable datagrams (not TCP)
- Client stub (name is produce)

```
int produce (int n)
{
    int status;
    send (sock, &n, sizeof(n));
    recv (sock, &status, sizeof(status));
    return(status);
}
```

- Server stub (name is arbitrary)

```
void produce_stub ()
{
    int n;
    int status;
    recv (sock, &n, sizeof(n));
    status = produce(n);
    send (sock, &status, sizeof(status));
    return(status);
}
```

- Client and server stubs can be generated automatically. What information do you need to generate a stub?

Problems with RPC

- RPC tries to make request/response to remote server look like a function call/return, but some differences remain

Making a distributed system look like a local system

- RPC: make request/response look like function call/return
- DSM: make multiple memories look like a single memory
- DFS: make disks on multiple computers look like a single file system
- Parallelizing compilers: make multiple CPUs look like one CPU
- Process migration (and RPC): allow users to easily use remote processors

Building distributed applications

- Why build distributed applications?
  - Performance
    - Aggregate performance of many computers can be faster than the performance of a single computer (even a fast one)
  - Reliability
    - Try to provide continuous service, even if some computers fail
    - Try to preserve data, even if some storage systems fail

Concurrency and distribution

- Distributed programs are multi-threaded, since each computer has at least one thread
- Need two mechanisms to write multi-threaded programs
  - Atomic primitive to synchronize threads
  - A way to share data between threads
- But neither of these mechanisms work for distributed applications
  - No shared memory

Send/receive as communication primitive

- No shared data; can only communicate information through message send/receive
- Can race conditions exist without shared data?

Send/receive as synchronization mechanism

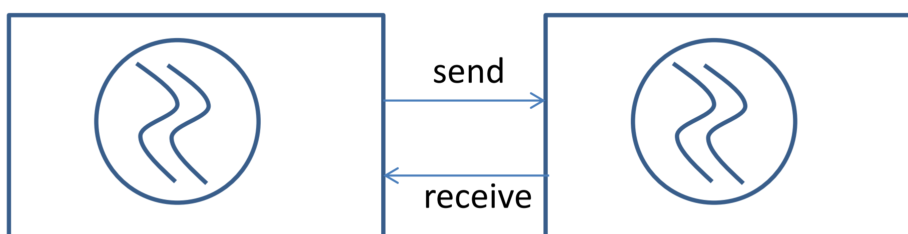
- What atomic primitive does the network hardware provide?
  - Send/receive a packet on the network
  - If packets A and B are sent to the same receiver, the receiver will receive A, B, or neither (but not a mixture of A and B)

- OS builds up from the hardware atomic primitive
  - If incoming packets from multi-packet messages are interleaved, the OS can separate the packets to form a whole message
  - OS ensures that a packet contains the correct data (e.g., not mixed up from several messages)

Structuring a concurrent system

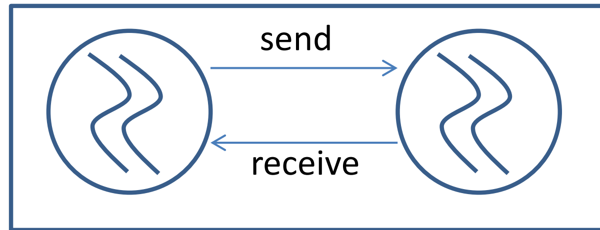
- One multi-threaded process on one computer

- Several multi-threaded process on several computers



Structuring a concurrent system

- Several multi-threaded process on each of several computers



- Why separate threads on one computer into separate address spaces, then use send/receive to communicate and synchronize?

- Microkernels
  - Operating system structure that separates OS functionality into several server processes, each in its own address space

Case study: distributed file systems

- Distributed file systems separates the storage of the data from the clients that use the data, but make it look as though all storage is local
- Benefits
  - Share data and resources between people
  - Enables uniform view from different computers
- Examples
  - AFS
  - SMB (Windows)
  - Dropbox
  - Google docs
  - WWW
- Basic implementation is simple
  - Send each request to server (classic client/server)
  - But poor performance. How to improve?

Client-side caching

- Assume many clients issuing requests to the file server
- Where will the bottleneck be?

- Use caching at clients to improve performance
  - Latency and throughput
  - Server scalability
  - Reduce network traffic
  - Availability?

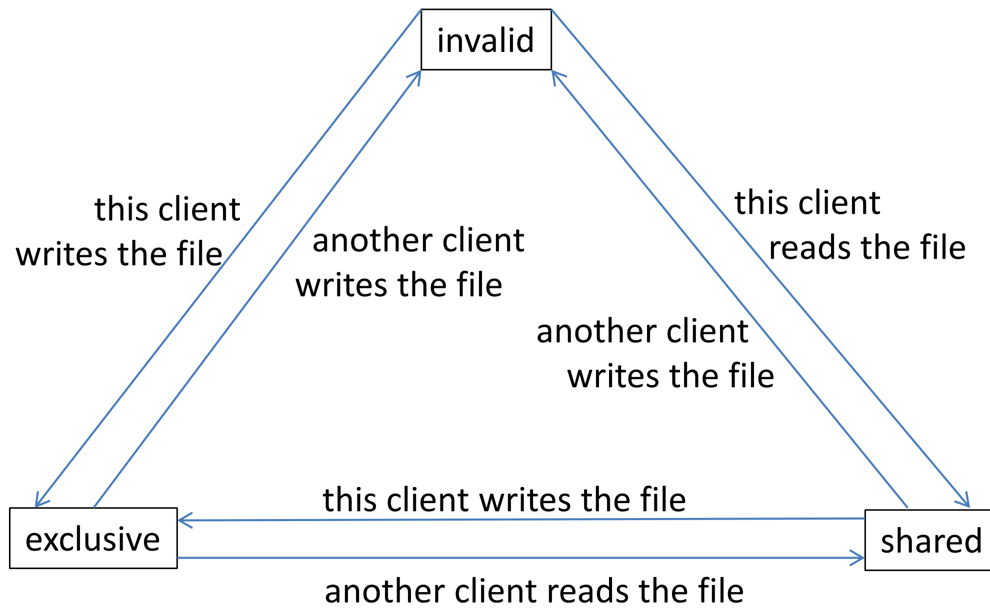
Client-side caching

- Implementation #1: transfer sole copy of file from server to client cache
  - Problems?

- Implementation #2: client cache stores a copy of the file
  - What happens if the client modifies its copy?

  - How to address the problem?

State of a client's copy of a file



Partitioning a file server across multiple servers

- To scale to a large number of clients, we can distribute the files across multiple file servers
- How to divide files between servers?

- Who knows which server holds a particular file?

Replication across multiple servers

- With more servers, likelihood of server failure increases
- Replication is the fundamental technique for tolerating failures
  - Store a file on multiple servers
  - E.g., primary + backup
    - Write data by writing to both primary and backup
    - Read data by reading from primary or backup
- What happens when a server fails?

- How to survive more than 1 failure (say, F failures)?

- Assumptions about how a server fails?

Fault models

- Fail stop
  - Machine stops executing
  - You can detect the failure
- Byzantine
  - Server may send erroneous messages
  - Server could be malicious

Byzantine generals

- E.g., 1 commander (C), 2 lietenants (L1, L2)
- Generals want to agree on a joint course of action (attack or retreat). All loyal generals must agree.
- Generals communicate with reliable (but unsigned messages). Recipient can tell who sent message (but not the true origin of that message).

- Solution #1
  - C sends to L1, L2
  - L1 and L2 follow C's orders
  - How can this break?

Byzantine generals

- Solution #2
  - L1, L2 send each other the message each got from C
  - All generals follow majority vote

  - E.g.,
    - C -> L1: attack
    - C -> L2: attack
    - L1 -> L2: attack
    - L2 -> L1: retreat

    - What should L1 do?

Byzantine generals

- Need 4 generals to handle 1 traitor (in general, needs 3F+1 generals to deal with F traitors)
- If L3 is traitor

  C -> L1: attack
  C -> L2: attack
  C -> L3: attack

  L1 -> L2: attack
  L1 -> L3: attack

  L2 -> L1: attack
  L2 -> L3: attack

  L3 -> L1: retreat
  L3 -> L2: retreat

  L1 and L2 each get 2 messages to attack and 1 message to retreat

Byzantine generals

- Need 4 generals to handle 1 traitor (in general, needs 3F+1 generals to deal with F traitors)
- If C is traitor

  C -> L1: attack
  C -> L2: attack
  C -> L3: retreat

  L1 -> L2: attack
  L1 -> L3: attack

  L2 -> L1: attack
  L2 -> L3: attack

  L3 -> L1: retreat
  L3 -> L2: retreat

  L1, L2, L3 each get 2 messages to attack and 1 message to retreat