Recall: 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!

Recall: 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 used to guarantee that all machines either commit or don’t

Recall: Network Protocols

- Networking protocols: many levels
  - Physical level: mechanical and electrical network (e.g., how are 0 and 1 represented)
  - Link level: packet formats/error control (for instance, the CSMA/CD protocol)
  - Network level: network routing, addressing
  - Transport Level: reliable message delivery

- Protocols on today’s Internet:
  - Ethernet
  - Wi-Fi
  - LTE
  - IP
  - UDP
  - TCP
  - NFS
  - RPC
  - WWW
  - e-mail
  - ssh
Network Layering

- **Layering:** building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the `Maximum Transfer Unit` or MTU: often 200-1500 bytes in size)
  - Routing is limited to within a physical link (wire) or perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

<table>
<thead>
<tr>
<th>Physical Reality: Packets</th>
<th>Abstraction: Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size (MTU)</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-machine</td>
<td>Process-to-process</td>
</tr>
<tr>
<td>Only on local area net</td>
<td>Routed anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Insecure</td>
<td>Secure</td>
</tr>
</tbody>
</table>

Recall: IPv4 Packet Format

- **IP Packet Format:**
  - IP Datagram: an unreliable, unordered, packet sent from source to destination
    - Function of network – deliver datagrams!

Physical Reality: Packets Abstraction: Messages

<table>
<thead>
<tr>
<th>Physical Reality</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size</td>
<td>Arbitrary</td>
</tr>
<tr>
<td>Unordered</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-machine</td>
<td>Process-to-process</td>
</tr>
<tr>
<td>Only on local area net</td>
<td>Routed anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Insecure</td>
<td>Secure</td>
</tr>
</tbody>
</table>

Building a messaging service on IP

- Process to process communication
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - Add "ports", which are 16-bit identifiers
    - A communication channel (connection) defined by 5 items:
      - [source addr, source port, dest addr, dest port, protocol]
- For example: The Unreliable Datagram Protocol (UDP)
  - Layered on top of basic IP (IP Protocol 17)
    - Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)
- Important aspect: low overhead!
  - Often used for high-bandwidth video streams
  - Many uses of UDP considered “anti-social” – none of the “well-behaved” aspects of (say) TCP/IP

Internet Architecture: Five Layers

- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
Internet Architecture: Five Layers

- Communication goes down to physical network
- Then from network peer to peer
- Then up to relevant layer

Layering Analogy: Packets in Envelopes

Internet Transport Protocols

- Datagram service (UDP): IP Protocol 17
  - No-frills extension of "best-effort" IP
  - Multiplexing/Demultiplexing among processes
- Reliable, in-order delivery (TCP): IP Protocol 6
  - Connection set-up & tear-down
  - Discarding corrupted packets (segments)
  - Retransmission of lost packets (segments)
  - Flow control
  - Congestion control
- Other examples:
  - DCCP (33), Datagram Congestion Control Protocol
  - RDP (26), Reliable Data Protocol
  - SCTP (132), Stream Control Transmission Protocol

Recall: Sockets in concept
Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    » If transmit close to maximum rate, get more throughput – even if some packets get lost
    » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    » Point-to-point network: insufficient queue at switch/router
    » Broadcast link: two hosts try to use same link
    » In any network: insufficient buffer space at destination
    » Rate mismatch: what if sender send faster than receiver can process?

- Reliable Message Delivery on top of Unreliable Packets
  - Need some way to make sure that packets actually make it to receiver
    » Every packet received at least once
    » Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

Transmission Control Protocol (TCP)

- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    » “Window” reflects storage at receiver – sender shouldn’t overrun receiver’s buffer space
    » Also, window should reflect speed/capacity of network – sender shouldn’t overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    » A “good citizen”

Problem: Dropped Packets

- All physical networks can garble or drop packets
  - Physical hardware problems (bad wire, bad signal)
- Therefore, IP can garble or drop packets
  - It doesn’t repair this itself (end-to-end principle!)
- Building reliable message delivery
  - Confirm that packets aren’t garbled
  - Confirm that packets arrive **exactly once**

Using Acknowledgements

- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending “ACK”) when packet received properly at destination
  - Timeout at sender: if no ACK, retransmit
- Some questions:
  - If the sender doesn’t get an ACK, does that mean the receiver didn’t get the original message?
    » No
  - What if ACK gets dropped? Or if message gets delayed?
    » Sender doesn’t get ACK, retransmits, Receiver gets message twice, ACK each
Stop-and-Wait (No Packet Loss)

- Send; wait for ACK; repeat
- Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back
  - One-way latency ($d$): one way delay from sender and receiver
- For symmetric latency, $RTT = 2d$

Stop-and-Wait (No Packet Loss)

- How fast can you send data?
- Little’s Law applied to the network: $n = B \cdot RTT$
- For Stop-and-Wait, $n = 1$ packet
- So bandwidth is 1 packet per RTT
  - Depends only on latency, not network capacity (!)

Stop-and-Wait (No Packet Loss)

- So bandwidth is 1 packet per RTT
  - Depends only on latency, not network capacity (!)
- Suppose RTT = 100 ms and 1 packet is 1500 bytes
- Throughput = $\frac{1500 \times 8}{0.1} = 120$ Kbps
- Very inefficient if we have a 100 Mbps link!

Stop-and-Wait with Packet Loss

- Loss recovery relies on timeouts
- How to choose a good timeout?
  - Too short – lots of duplication
  - Too long – packet loss is really disruptive!
- How to deal with duplication?
  - Retransmission certainly opens up the possibility for
How to Deal with Message Duplication?

- **Solution**: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate number’s; Discard if detected
- **Requirements**:
  - Sender keeps copy of unACK’d messages
    - Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - Hard: when ok to forget about received message?
- **Alternating-bit protocol**:
  - Send one message at a time; don’t send next message until ACK received
  - Sender keeps last message; receiver tracks sequence number of last message received
- **Pros**: simple, small overhead
- **Cons**: doesn’t work if network can delay or duplicate messages arbitrarily

Advantages of Moving Away From Stop-and-Wait

- **Larger space of acknowledgements**
  - Pipelining: don’t wait for ACK before sending next packet
- **ACKs serve dual purpose**:
  - Reliability: Confirming packet received
  - Ordering: Packets can be reordered at destination
- **How much data is in flight now?**
  - Bytes in-flight: \( W_{\text{send}} = RTT \times B \)
  - Here B is in “bytes/second”
  - \( W_{\text{send}} \) = Sender’s “Window Size”
  - Packets in flight = \( W_{\text{send}} / \text{packet size} \)
- **How long does the sender have to keep the packets around?**
- **How long does the receiver have to keep the packets’ data?**
- **What if sender is sending packets faster than the receiver can process the data?**

Administrivia

- **Midterm 3**: Thursday 12/3: 5-7PM as before
  - Material up to Lecture 25
  - Cameras and Zoom screen sharing again as with Midterm 2
  - Review session TBA
- **Lecture 26** will be a fun lecture
  - Let me know if there are topics you would like to discuss!

Recall: CS 162 Collaboration Policy

- **Explaining a concept to someone in another group**
- **Discussing algorithms/testing strategies with other groups**
- **Discussing debugging approaches with other groups**
- **Searching online for generic algorithms (e.g., hash table)**
- **Sharing code or test cases with another group**
- **Copying OR reading another group’s code or test cases**
- **Copying OR reading online code or test cases from prior years**
- **Helping someone in another group to debug their code**

- **We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders**
- **Don’t put a friend in a bad position by asking for help that they shouldn’t give!**
Recall: Communication Between Processes

- Data written by A is held in memory until B reads it
- Queue has a fixed capacity
  - Writing to the queue blocks if the queue is full
  - Reading from the queue blocks if the queue is empty
- POSIX provides this abstraction in the form of pipes

```c
write(wfd, wbuf, wlen);
```

```c
n = read(rfd, rbuf, rmax);
```

Buffering in a TCP Connection

- A single TCP connection needs four in-memory queues:
  - Send buffer: add data on write syscall, remove data when ACK received
  - Receive buffer: add data when packets received, remove data on read syscall

Window Size: Space in Receive Queue

- A host's window size for a TCP connection is how much remaining space it has in its receive queue
- A host advertises its window size in every TCP packet it sends!
- Sender never sends more than receiver’s advertised window size

Sliding Window Protocol

- TCP sender knows receiver’s window size, and aims never to exceed it
- But packets that it previously send may arrive, filling the window size!

**Rule:** TCP sender ensures that:

- Number of Sent but UnACKed Bytes < Receiver’s Advertised Window Size

- Can send new packets as long as sent-but-unacked packets haven’t already filled the advertised window size
Sliding Window (No Packet Loss)

- Example: Window size ($w$) = 3 packets
- Window size to fill link is given by: $w = B_{\text{pkt}} \cdot \text{RTT}$
- $B_{\text{pkt}}$ = Packets/sec
- Little's Law once again!

- For TCP, window is in bytes, not packets

TCP Windows and Sequence Numbers: PER BYTE!

- Sender has three regions:
  - Sequence regions
    - sent and ACK'd
    - sent and not ACK'd
    - not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - received and ACK'd (given to application)
    - received and buffered
    - not yet received (or discarded because out of order)

Window-Based Acknowledgements (TCP)

Congestion

- Too much data trying to flow through some part of the network
- IP's solution: Drop packets
- What happens to TCP connection?
  - Lots of retransmission – wasted work and wasted bandwidth (when bandwidth is scarce)
**Congestion Avoidance**

- **Congestion**
  - How long should timeout be for re-sending messages?
    - Too long → wastes time if message lost
    - Too short → retransmit even though ACK will arrive shortly
  - Stability problem: more congestion ⇒ ACK is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - Closely related to window size at sender: too big means putting too much data into network

- **How does the sender's window size get chosen?**
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of $N$
    - Goal: fill network between sender and receiver
    - Basic technique: slowly increase size of window until acknowledgements start being delayed/lost

- **TCP solution: "slow start" (start sending slowly)**
  - If no timeout, slowly increase window size (throughput) by 1 for each ACK received
  - Timeout ⇒ congestion, so cut window size in half
  - "Additive Increase, Multiplicative Decrease"

**Congestion Management**

- TCP artificially restricts the window size if it sees packet loss
- Careful control loop to make sure:
  1. We don't send too fast and overwhelm the network
  2. We utilize most of the bandwidth the network has available
     - In general, these are conflicting goals!

---

**Recall: Connection Setup over TCP/IP**

- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

- 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

**Establishing TCP Service**

1. Open connection: 3-way handshaking
2. Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
   - Indication if connection fails: Reset
3. Close (tear-down) connection
Sockets in concept

Client
- Create Client Socket
- Connect it to server (host:port)
- Close Client Socket

Server
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall
- Close Server Socket

Open Connection: 3-Way Handshake

Client (initiator)
- connect()
- SYN, SeqNum = x
- SYN and ACK, SeqNum = y and Ack = x + 1
- ACK, Ack = y + 1

Server
- listen()

Close Connection: 4-Way Teardown

Host 1
- close()
- FIN
- OS deallocates connection state
- Any calls to read() return 0

Host 2
- FIN
- Host 1 can retransmit FIN ACK if it is lost
- FIN ACK
- OS deallocates connection state
- OS discards data (no socket to give it to)

- Connection is not closed until both sides agree
- If multiple FDs on Host 1 refer to this connection, all of them must be closed
- Same for close() call on Host 2
Recall: Distributed Applications Build With Messages

- 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

Question: Data Representation

- 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

```c
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 is the most-significant bits
  • Little Endian: address is the least-significant bits

<table>
<thead>
<tr>
<th>Processor</th>
<th>Endianness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>B (Big/Little Endian)</td>
</tr>
<tr>
<td>AXP</td>
<td>B (Big/Little Endian)</td>
</tr>
<tr>
<td>MIPS</td>
<td>B (Big/Little Endian)</td>
</tr>
<tr>
<td>PowerPC (PPC)</td>
<td>B (Big Endian)</td>
</tr>
<tr>
<td>Sun Sparc</td>
<td>B (Big Endian)</td>
</tr>
<tr>
<td>Intel 80386</td>
<td>B (Big Endian)</td>
</tr>
<tr>
<td>Intel x86 (32 bit)</td>
<td>L (Little Endian)</td>
</tr>
<tr>
<td>Intel x86_64 (64 bit)</td>
<td>L (Little Endian)</td>
</tr>
</tbody>
</table>

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 Endian is the Internet?

• Big Endian
  • Network byte order
  • Vs. “host byte order”

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?
Data Serialization Formats

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

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");`
Client (caller) 
\[ r = f(v_1, v_2); \]

Server (callee) 
\[ res_t f(a_1, a_2) \]

**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.

**RPC Details (1/3)**

- Equivalence with regular procedure call
  - Parameters \(\iff\) Request Message
  - Result \(\iff\) 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 \( \rightarrow \) 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 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 \( \prec \) same-machine RPC \( \prec \) 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.

Network-Attached Storage and the CAP Theorem

- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same time
  - Otherwise known as “Brewer’s Theorem”

Summary

- **TCP**: Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network
- **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
- **Distributed File System**:
  - Transparent access to files stored on a remote disk
  - Caching for performance