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.

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 this 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 losy 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.
- 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.

Goals of Today’s Lecture

- TCP flow control
- Two-Phase Commit
- RPCs

Flow Control

- Recall: Flow control ensures a fast sender does not overwhelm a slow receiver.
- Example: Producer-consumer with bounded buffer (Lecture 5)
  - A buffer between producer and consumer
  - Producer puts items into buffer as long as buffer not full
  - Consumer consumes items from buffer.

![Diagram of producer-consumer with bounded buffer]
TCP Flow Control

- TCP: sliding window protocol at byte (not packet) level
  - Go-back-N: TCP Tahoe, Reno, New Reno
  - Selective Repeat (SR): TCP Sack

- Receiver tells sender how many more bytes it can receive without overflowing its buffer (i.e., AdvertisedWindow)

- The ack(nowledgement) contains sequence number N of next byte the receiver expects, i.e., receiver has received all bytes in sequence up to and including N-1

TCP Flow Control

- TCP/IP implemented by OS (Kernel)
  - Cannot do context switching on sending/receiving every packet
    - At 1Gbps, it takes 12 usec to send an 1500 bytes, and 0.8usec to send an 100 byte packet

- Need buffers to match …
  - sending app with sending TCP
  - receiving TCP with receiving app

TCP Flow Control

- Three pairs of producer-consumer’s
  ① sending process → sending TCP
  ② Sending TCP → receiving TCP
  ③ receiving TCP → receiving process

TCP Flow Control

- Example assumptions:
  - Maximum IP packet size = 100 bytes
  - Size of the receiving buffer (MaxRcvBuF) = 300 bytes

- Recall, ack indicates the next expected byte in-sequence, not the last received byte
- Use circular buffers
Circular Buffer

- Assume
  - A buffer of size N
  - A stream of bytes, where bytes have increasing sequence numbers
    » Think of stream as an unbounded array of bytes and of sequence number as indexes in this array
- Buffer stores at most N consecutive bytes from the stream
- Byte \( k \) stored at position \( (k \mod N) + 1 \) in the buffer

**Buffered data**

sequence #: [27, 28, 29, 30, 31, 32, 33, 34, 35, 36]

Circular buffer: (N = 10)

- Start
- End

TCP Flow Control

- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver
- LastByteRcvd: last byte received by receiver from sender
- NextByteExpected: last in-sequence byte expected by receiver
- LastByteRead: last byte read by the receiving process

**Receiving Process**

- NextByteExpected
- LastByteRcvd
- LastByteRead

**Sending Process**

- LastByteWritten
- MaxSendBuffer
- LastByteSent
- LastByteAcked

**AdvertisedWindow**

\[ \text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \]

**SenderWindow**

\[ \text{SenderWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked}) \]

**WriteWindow**

\[ \text{WriteWindow} = \text{MaxSendBuffer} - (\text{LastByteWritten} - \text{LastByteAcked}) \]

TCP Flow Control

- Still true if receiver missed data....

- AdvertisedWindow: number of bytes TCP receiver can receive

**AdvertisedWindow**

\[ \text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \]

**WriteWindow**: number of bytes sending process can write

\[ \text{WriteWindow} = \text{MaxSendBuffer} - (\text{LastByteWritten} - \text{LastByteAcked}) \]
**TCP Flow Control**

- Sending app sends 350 bytes
- Recall:
  - We assume IP only accepts packets no larger than 100 bytes
  - MaxRcvBuf = 300 bytes, so initial Advertised Window = 300 bytes

---

**TCP Flow Control**

- Sending app sends 1st packet (i.e., first 100 bytes) and receiver gets the packet
  - Ack=101, AdvWin = 200

---

**TCP Flow Control**

- Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet
  - Ack=101, AdvWin = 200
TCP Flow Control

**Sending Process**
- `LastByteWritten(350)`
- `LastByteSent(200)`
- `Data[101,200]`
- `Data[1,200]`
- `Ack=101, AdvWin = 200`

**Receiving Process**
- `LastByteRead(0)`
- `LastByteAcked(0)`
- `LastByteRcvd(200)`
- `NextByteExpected(201)`
- `Data[1,100]`
- `Data[1,200]`

**Sender sends 2<sup>nd</sup> packet (i.e., next 100 bytes) and receiver gets the packet**

**TCP Flow Control**

**Sending Process**
- `LastByteWritten(350)`
- `LastByteSent(200)`
- `Data[101,200]`
- `Data[1,200]`
- `Ack=101, AdvWin = 200`

**Receiving Process**
- `LastByteRead(100)`
- `LastByteAcked(0)`
- `LastByteRcvd(100)`
- `NextByteExpected(201)`
- `Data[1,100]`
- `Data[1,200]`

**Receiving TCP delivers first 100 bytes to receiving process**

**TCP Flow Control**

**Sending Process**
- `LastByteWritten(350)`
- `LastByteSent(300)`
- `Data[201,300]`
- `Data[1,300]`
- `Ack=201, AdvWin = 200`

**Receiving Process**
- `LastByteRead(100)`
- `LastByteAcked(0)`
- `LastByteRcvd(300)`
- `NextByteExpected(201)`
- `Data[1,100]`
- `Data[1,200]`

**Sender sends 3<sup>rd</sup> packet (i.e., next 100 bytes) and the packet is lost**

**AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead) = 300 – (200 – 100) = 200**
TCP Flow Control

**Sending Process**
- LastByteWritten(350)
- LastByteSent(300)
- LastByteAcked(0)

**Receiving Process**
- LastByteRcvd(200)
- NextByteExpected(201)
- LastByteRead(100)

Sender stops sending as window full

\[
SndWin = AdvWin - (LastByteSent - LastByteAcked)
\]

\[
= 300 - (300 - 0) = 0
\]

- Ack for 1st packet (ack indicates next byte expected by receiver)
- Receiver no longer needs first 100 bytes

\[
SndWin = AdvWin - (LastByteSent - LastByteAcked)
\]

\[
= 200 - (300 - 100) = 0
\]
TCP Flow Control

- **Sending Process**
  - LastByteWritten(350)
  - LastByteAcked(200)
  - LastByteSent(350)

- **Receiving Process**
  - LastByteRead(100)
  - LastByteRcvd(100)
  - LastByteSent(350)

- Data\([1,100]\)
- Data\([1,200]\)
- Data\([1,300]\)
- Data\([101,300]\)
- Data\([201,300]\)

- **Receiver gets ack for 2nd packet**
- **AdvWin = 200 bytes**

- SndWin = AdvWin = 200 bytes

- **NextByteExpected(201)**

- **Ack=201, AdvWin = 200**

- Sender can now send new data!

- \(SndWin = AdvWin - (LastByteSent - LastByteAcked) = 100\)
TCP Flow Control

Sending Process

LastByteWritten(350)

{201, 301, 350}

LastByteSent(350)

{201, 301, 350}

LastByteAcked(200)

{201, 301, 350}

LastByteWritten(350)

{201, 301, 350}

LastByteRcvd(350)

{201, 301, 350}

NextByteExpected(201)

{201, 301, 350}

Receiving Process

LastByteAcked(200)

{201, 301, 350}

LastByteRead(100)

{201, 301, 350}

LastByteSent(350)

{201, 301, 350}

LastByteRcvd(350)

{201, 301, 350}

NextByteExpected(201)

{201, 301, 350}

Data[301,350]

{[201,350]}

{$[101,200],[301,350]$}

• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 3\textsuperscript{rd} packet?

Yes! Sender can re-send 2\textsuperscript{nd} packet since it's in existing window
– won't cause receiver window to grow

Data[201,300]

{[201,350]}

{$[101,300]$}

Data[301,350]

{[201,350]}

{$[101,301]$}
TCP Flow Control

**Sending Process**
- `LastByteWritten(350)`
- `LastByteAcked(200)`
- `LastByteSent(201, 301, 350)`
- `Data[301, 350]`
- `NextByteExpected[351]`
- `Ack=201, AdvWin = 50`
- `Data[201, 300]`
- `Ack=351, AdvWin = 50`
- `Sender gets 3rd packet and sends Ack for 351`
- `AdvWin = 50`

**Receiving Process**
- `LastByteRead(100)`
- `LastByteRecvd(350)`
- `LastByteWritten(350)`
- `LastByteRcvd(350)`
- `NextByteExpected[351]`
- `Data[301, 350]`
- `Ack=201, AdvWin = 50`
- `Data[201, 300]`
- `Ack=351, AdvWin = 50`

**Discussion**
- Why not have a huge buffer at the receiver (memory is cheap)?
- Sending window (SndWnd) also depends on network congestion
  - **Congestion control**: ensure that a fast receiver doesn’t overwhelm a router in the network (discussed in detail in cs168)
- In practice there is another set of buffers in the protocol stack, at the **link layer** (i.e., Network Interface Card)

**Administrivia**
- Midterm 3 coming up on **Mon 4/24 6:30-8PM**
  - All topics up to and including Lecture 15
    - Focus will be on Lectures 16 – 23 and associated readings, and Projects 3
    - But expect 20-30% questions from materials from Lectures 1-15
  - VLSB 2040 and VLSB 2060
  - Closed book
  - 2 pages hand-written notes both sides
**Goals of Today’s Lecture**

- TCP flow control
- Two-Phase Commit
- RPCs

---

**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**

- 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!

---

**BREAK**
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

• Two-Phase Commit protocol: Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)

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

2PC Algorithm

• One coordinator
• N workers (replicas)
• High level algorithm description:
  – Coordinator asks all workers if they can commit
  – If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT"
  – Otherwise coordinator broadcasts "GLOBAL-ABORT"
  – Workers obey the GLOBAL messages
• Use a persistent, stable log on each machine to keep track of what you are doing
  – If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash

Detailed Algorithm

Coordinator Algorithm

Worker Algorithm

Coordinator sends VOTE-REQ to all workers

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

– If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers

– If doesn’t receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

– If receive GLOBAL-COMMIT then commit
– If receive GLOBAL-ABORT then abort
Failure Free Example Execution

State Machine of Coordinator

- Coordinator implements simple state machine:

  - INIT
  - WAIT
  - ABORT
  - COMMIT

  - Recv: START
  - Send: VOTE-REQ
  - Recv: all VOTE-COMMIT
  - Send: GLOBAL-COMMIT

  - Recv: VOTE-ABORT
  - Send: GLOBAL-ABORT

Dealing with Worker Failures

- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in "WAIT" state
- In WAIT, if doesn’t receive N votes, it times out and sends GLOBAL-ABORT
Example of Worker Failure

- **INIT**
- **WAIT**
- **ABORT**
- **COMM**
- **GLOBAL-ABORT**

- **COORDINATOR**
- **WORKER 1**
- **WORKER 2**
- **WORKER 3**

Example of Coordinator Failure #1

- **INIT**
- **READY**
- **ABORT**
- **COMM**

Dealing with Coordinator Failure

- Worker waits for **VOTE-REQ** in **INIT**
  - Worker can time out and abort (coordinator handles it)
- Worker waits for **GLOBAL-*** message in **READY**
  - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send **GLOBAL-*** message

Example of Coordinator Failure #2

- **INIT**
- **READY**
- **ABORT**
- **COMM**

Example of Coordinator Failure #1

- **INIT**
- **READY**
- **ABORT**
- **COMM**

Example of Coordinator Failure #2

- **COORDINATOR**
- **WORKER 1**
- **WORKER 2**
- **WORKER 3**

4/17/2017 CS162 ©UCB Spring 2017 Lec 21.53

4/17/2017 CS162 ©UCB Spring 2017 Lec 21.54

4/17/2017 CS162 ©UCB Spring 2017 Lec 21.55

4/17/2017 CS162 ©UCB Spring 2017 Lec 21.56
Durability

- All nodes use stable storage to store current state
  - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.

- Upon recovery, it can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker commits in COMMIT
  - Worker asks Coordinator in READY

Blocking for Coordinator to Recover

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

Distributed Decision Making Discussion (1/2)

- 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

Distributed Decision Making Discussion (2/2)

- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    » Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes
    » Site A crashes
    » Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
    » B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
PAXOS

- PAXOS: An alternative used by Google and others that does not have this blocking problem
  - Developed by Leslie Lamport (Turing Award Winner)
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

Byzantine General's Problem

- Byzantine General's Problem (n players):
  - One General 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

Goals of Today's Lecture

- TCP flow control
- Two-Phase Commit
- RPCs
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
- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    \[ \text{remoteFileSystem} \rightarrow \text{Read}(\text{"rutabaga"}); \]
  - Translated automatically into call on server:
    \[ \text{fileSys} \rightarrow \text{Read}(\text{"rutabaga"}); \]

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 Information Flow

RPC Details (1/3)

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    » Code for server to unpack message, call procedure, pack results, send them off
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 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

RP 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

- Cost of Procedure call « same-machine RPC « network RPC
- Means programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex
Summary

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

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

• Remote Procedure Call (RPC): Call procedure on remote machine
  – Provides same interface as procedure
  – Automatic packing/unpacking of args without user programming