Consistency

David E. Culler
CS162 – Operating Systems and Systems Programming
http://cs162.eecs.berkeley.edu/
Lecture 35
Nov 19, 2014
Recap: TCP Flow Control

Sending Process:
- LastByteWritten(350)
- LastByteAced(200)
- LastByteSent(350)

Receiving Process:
- LastByteRead(100)
- LastByteRcvd(350)
- NextByteExpected(201)

Data:
- Data[1,100] = {1,100}
- Data[101,200] = {101, 200}
- Data[301,350] = {201, 350}

AdvertisenedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

SenderWindow = AdvertisenedWindow – (LastByteSent – LastByteAced)

WriteWindow = MaxSendBuffer – (LastByteWritten – LastByteAced)

Ack = 201, AdvWin = 50
Summary: Reliability & Flow Control

- Flow control: three pairs of producer consumers
  - Sending process $\rightarrow$ sending TCP
  - Sending TCP $\rightarrow$ receiving TCP
  - Receiving TCP $\rightarrow$ receiving process

- AdvertisedWindow: tells sender how much new data the receiver can buffer

- SenderWindow: specifies how more the sender can transmit.
  - Depends on AdvertisedWindow and on data sent since sender received AdvertisedWindow

- WriteWindow: How much more the sending application can send to the sending OS
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 sender doesn’t overwhelm a router in the network
  – discussed in detail in CS168

• In practice there is other sets of buffers in the protocol stack, at the link layer (i.e., Network Interface Card)
Internet Layering – engineering for intelligence and change

**Application Layer**
- Data

**Transport Layer**
- Data
- **Trans.Hdr.**

**Network Layer**
- Data
- **Net.Hdr.**
- **Trans.Hdr.**

**Datalink Layer**
- Data
- **FrameHdr.**
- **Net.Hdr.**
- **Trans.Hdr.**

**Physical Layer**
- **101010100110101110**

Any distributed protocol (e.g., HTTP, Skype, p2p, KV protocol in your project)

Send **segments** to another **process** running on same or different node

Send **packets** to another node possibly **located** in a different network

Send **frames** to other node directly connected to same physical network

Send **bits** to other node directly connected to same physical network
The Shared Storage Abstraction

- Information (and therefore control) is communicated from one point of computation to another by:
  - The former storing/writing/sending to a location in a shared address space
  - And the second later loading/reading/receiving the contents of that location

- Memory (address) space of a process
- File systems
- Dropbox, ...
- Google Docs, ...
- Facebook, ...
What are you assuming?

• **Writes happen**
  – Eventually a write will become visible to readers
  – Until another write happens to that location

• **Within a sequential thread, a read following a write returns the value written by that write**
  – Dependences are respected
  – Here a control dependence
  – Each read returns the most recent value written to the location
For example

Write: \( A := 162 \)

Read: \( \text{print}(A) \)

Read: \( \text{print}(A) \)
What are you assuming?

• **Writes happen**
  – Eventually a write will become visible to readers
  – Until another write happens to that location

• **Within a sequential thread, a read following a write returns the value written by that write**
  – Dependences are respected
  – Here a control dependence
  – Each read returns the most recent value written to the location

• **A sequence of writes will be visible in order**
  – Control dependences
  – Data dependences
For example

Write: \( A := 162 \)

Read: print(A)

Read: print(A)

Write: \( A := A + 1 \)

162, 163, 170, 164, 171, ...

162, 163, 170, 171, ...

162, 163, 170, 171, ...

X
What are you assuming?

- **Writes happen**
  - Eventually a write will become visible to readers
  - Until another write happens to that location

- **Within a sequential thread, a read following a write returns the value written by that write**
  - Dependences are respected
  - Here a control dependence
  - Each read returns the most recent value written to the location

- **A sequence of writes will be visible in order**
  - Control dependences
  - Data dependences
  - May not see every write, but the ones seen are consistent with order written

- **A readers see a consistent order**
  - It is as if the total order was visible to all and they took samples
For example

Write: \( A := 162 \)

Read: print(A)

Write: \( A := A + 1 \)

Read: print(A)

162, 163, 170, 171, ...

Read: print(A)

164, 170, 186, ...

Read: print(A)
Demo

- https://docs.google.com/a/berkeley.edu/spreadsheets/d/1INjjYqUnFurPLKnnWrexx09Ww5LS5BhNxKt3BoJY6Eg/edit
For example

A := 162

Write: A := 199

Write: A := 61

Read: print(A)

162, 199, 199, 61, 61 ...
162, 61, 199, ...
61, 199, ...
162, 199, 61, 199 ...
162, 199, 61, 199 ...
For example

A := 162
Write: A := 199
Read: print(A)
162, 199, 199, 61, 61, ...

Write: A := 61
Read: print(A)
162, 199, 61, ...

162, 61, ...
162, 61, 199, ...

162, 199, 199, 61, 61, ...
162, 199, 61, ...
162, 61, ...
162, 61, 199, ...

162, 199, 199, 61, 61, ...
What is the key to performance AND reliability

• Replication
What is the source of inconsistency?

• Replication
Any Storage Abstraction
Multiple Clients access server: OK

- But slow
Multi-level Storage Hierarchy: OK

- Replication within storage hierarchy to make it fast
Multiple Clients and Multi-Level

- Fast, but not OK
Multiple Servers

- What happens if cannot update all the replicas?
- Availability => Inconsistency
Basic solution to multiple client replicas

- Enforce single-writer multiple reader discipline
- Allow readers to cache copies
- Before an update is performed, writer must gain exclusive access
- Simple Approach: invalidate all the copies then update
- Who keeps track of what?
The Multi-processor/Core case

- Interconnect is a broadcast medium
- All clients can observe all writes and invalidate local replicas (write-thru invalidate protocol)
• **Write-Back via read-exclusive**
• **Atomic Read-modify-write**
NFS “Eventual” Consistency

- Stateless server allows multiple cached copies
  - Files written locally (at own risk)
- Update Visibility by “flush on close”
- GetAttributes on file ops to check modify since cache
Other Options

• Server can keep a “directory” of cached copies
• On update, sends invalidate to clients holding copies
• Or can send updates to clients
• Pros and Cons ???

• OS Consistency = Architecture Coherence requires invalidate copies prior to write
• Write buffer has be to be treated as primary copy
  – like transaction log
Multiple Servers

- What happens if cannot update all the replicas?
- Availability => Inconsistency
Durability and Atomicity

• How do you make sure transaction results persist in the face of failures (e.g., server node failures)?

• Replicate store / database
  – Commit transaction to each replica

• What happens if you have failures during a transaction commit?
  – Need to ensure atomicity: either transaction is committed on all replicas or none at all
Two Phase (2PC) Commit

• 2PC is a distributed protocol

• High-level problem statement
  – If no node fails and all nodes are ready to commit, then all nodes COMMIT
  – Otherwise ABORT at all nodes

• Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
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
Coordinator Algorithm

Coordinator sends VOTE-REQ to all workers

Worker Algorithm

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

- 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

coordinator

worker 1

worker 2

worker 3

VOTE-REQ

GLOBAL-COMMIT

VOTE-COMMIT

time
State Machine of Coordinator

- Coordinator implements simple state machine

```
INIT
  Recv: START
  Send: VOTE-REQ

WAIT
  Recv: VOTE-ABORT
  Send: GLOBAL-ABORT
  Recv: VOTE-COMMIT
  Send: GLOBAL-COMMIT

ABORT

COMMIT
```
State Machine of Workers

- **INIT**
  - Receive: VOTE-REQ
  - Send: VOTE-ABORT

- **READY**
  - Receive: VOTE-REQ
  - Send: VOTE-COMMIT

- **ABORT**
  - Receive: GLOBAL-ABORT

- **COMMIT**
  - Receive: GLOBAL-COMMIT
Dealing with Worker Failures

• How to deal with worker failures?
  – Failure only affects states in which the node 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

coordinator

worker 1

worker 2

worker 3

INIT

WAIT

ABORT

COMM

VOTE-REQ

VOTE-COMMIT

GLOBAL-ABORT

time

timeout

X
Dealing with Coordinator Failure

How to deal with coordinator failures?

- 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

```
Recv: VOTE-REQ
Send: VOTE-ABORT
```

```
Recv: VOTE-REQ
Send: VOTE-COMMIT
```

```
Recv: GLOBAL-ABORT
```

```
Recv: GLOBAL-COMMIT
```

INIT

READY

ABORT

COMMIT
Example of Coordinator Failure #1

- Coordinator
- Worker 1
- Worker 2
- Worker 3

Flowchart:
- INIT
- READY
- ABORT
- COMM

VOTE-REQ
timeout
VOTE-ABORT
timeout
timeout
Example of Coordinator Failure #2

Coordinator

Worker 1

Worker 2

Worker 3

VOTE-REQ

VOTE-COMMIT

GLOBAL-ABORT

Block waiting for coordinator

Restarted

VOTE-REQ

VOTE-COMMIT

GLOBAL-ABORT

Block waiting for coordinator

INIT

READY

ABORT

COMM

Coordinator

Worker 1

Worker 2

Worker 3

VOTE-REQ

VOTE-COMMIT

GLOBAL-ABORT

Block waiting for coordinator

Restarted
Durability

- All nodes use stable storage* to store which state they are in

- 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

* - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
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)