Networks, Protocols and Distributed Systems

A Slightly Theoretic Crash Course

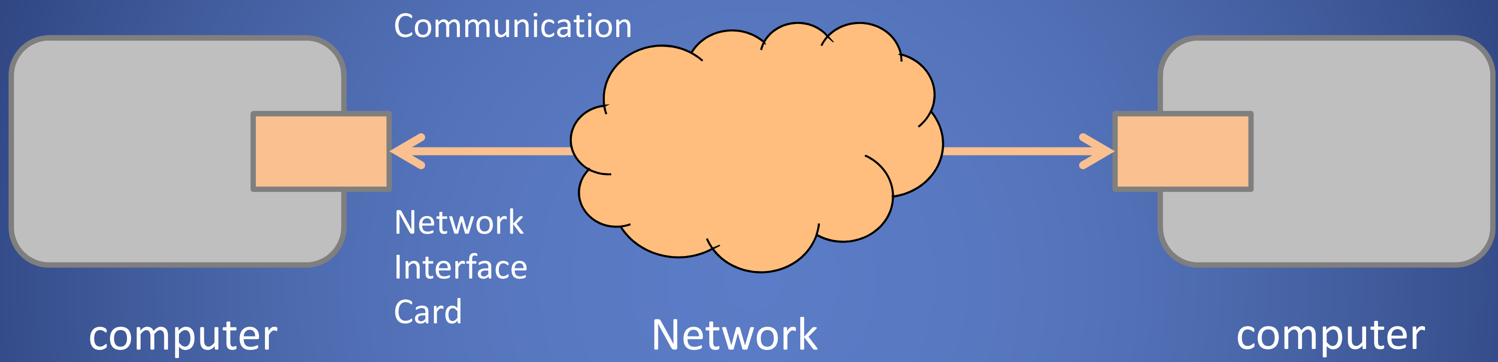
Haraldur Darri Þorvaldsson

Overview of this Talk

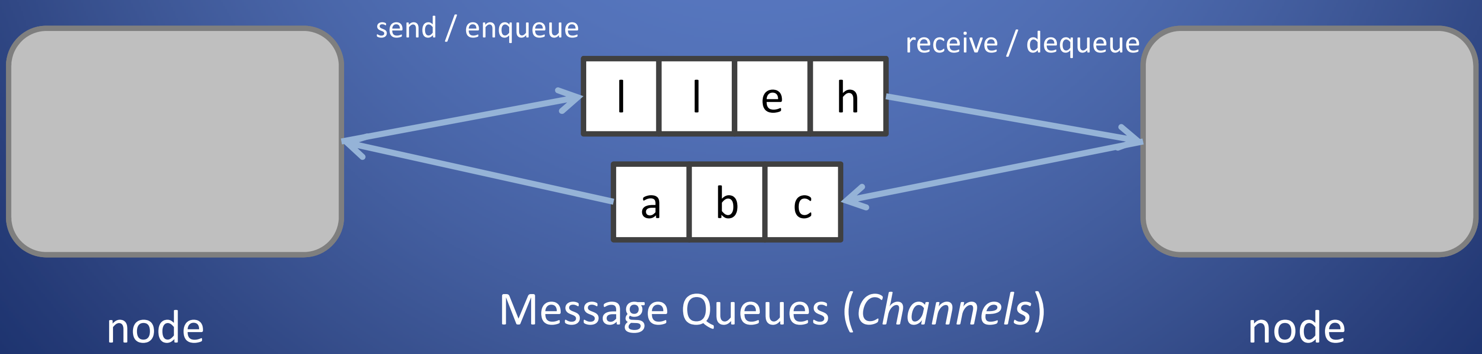
- Networks as graphs of queues
- Blocking / Non-Blocking program styles
- Reliable / Unreliable network channels
- Concrete examples: TCP, UDP
- MMO’s Abstracted: Shared Distributed State
- Wider applicability of network model

Networks as Graphs of Queues

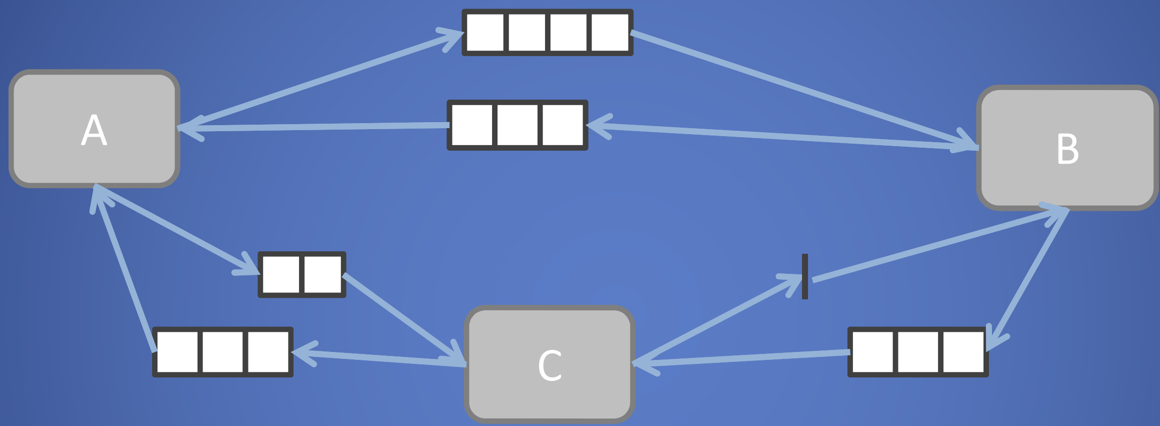
Typical Diagram View: some abstractions, a dash of hardware …



Today: Programmer’s View / Model: *Queues* of *Messages*



The Basic Distributed Systems Model



- A bunch of *nodes* exchanging *messages* across dedicated *channels*: pairs of uni-directional queues
- *Nodes cannot observe or modify other nodes directly*
  - *All* inter-node effects are through messages

The Life of a Node

- A node has a sequence of *events*, which can be:
  1. A *computation* step (changing node's *state*)
     - Basically: the sequential execution of a program snippet
  2. A *send* event (enqueues a msg on a channel)
  3. A *receive* event (dequeues a msg from a channel)
- A message contains a finite amount of data
  - For example: a string over some alphabet
    - Physical messages (*packets*) typically 50-9000 bytes
- No model of time; only sequences of events

The Life of a Channel

- When a message is enqueued to a channel:
  – Appends message to end of its queue
- When channel asked to dequeue a message:
  – Removes and *deliver* msg at front of its queue
- This describes a “perfect” *reliable channel*
  – Real networks fail, we mitigate with clever software as much as possible
- Example: Transmission Control Protocol (TCP)
  – Delivers the correct bytestream (if anything)

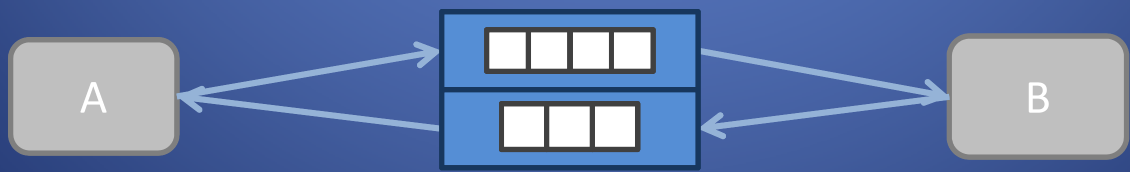
Distributed Algorithm / Protocol Simple Example: Load Balancing

Two nodes execute the following pseudo-code:

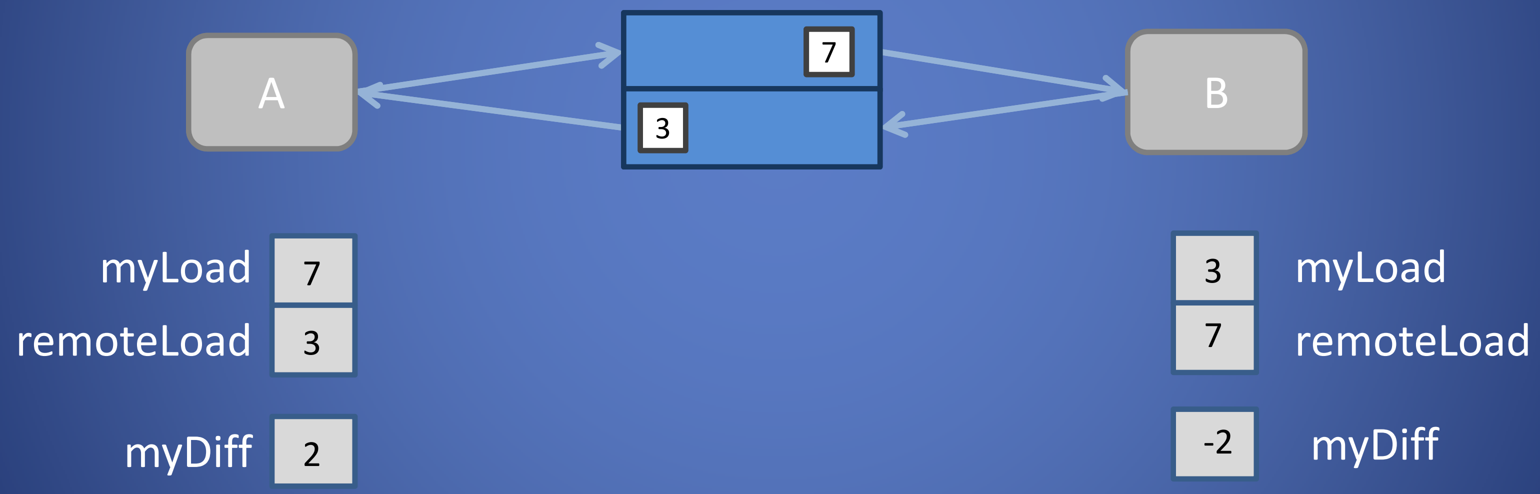
Computation events

```
myLoad = ComputeCurrentLoad()
send(myLoad)
remoteLoad = receive()
halfLoad = (myLoad + remoteLoad) / 2
if myLoad > halfLoad:
   hand off (myLoad – halfLoad) units of work
else if myLoad < halfLoad:
   take on (halfLoad – myLoad) units of work
```

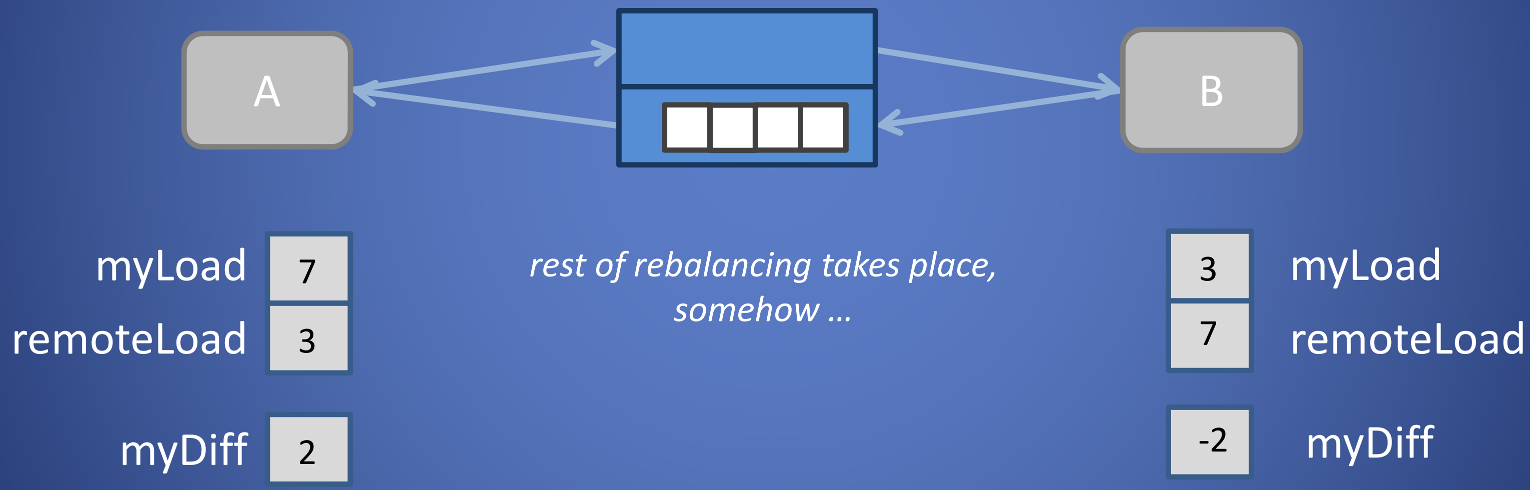
Networking events



Animimated of Algorithm Instance

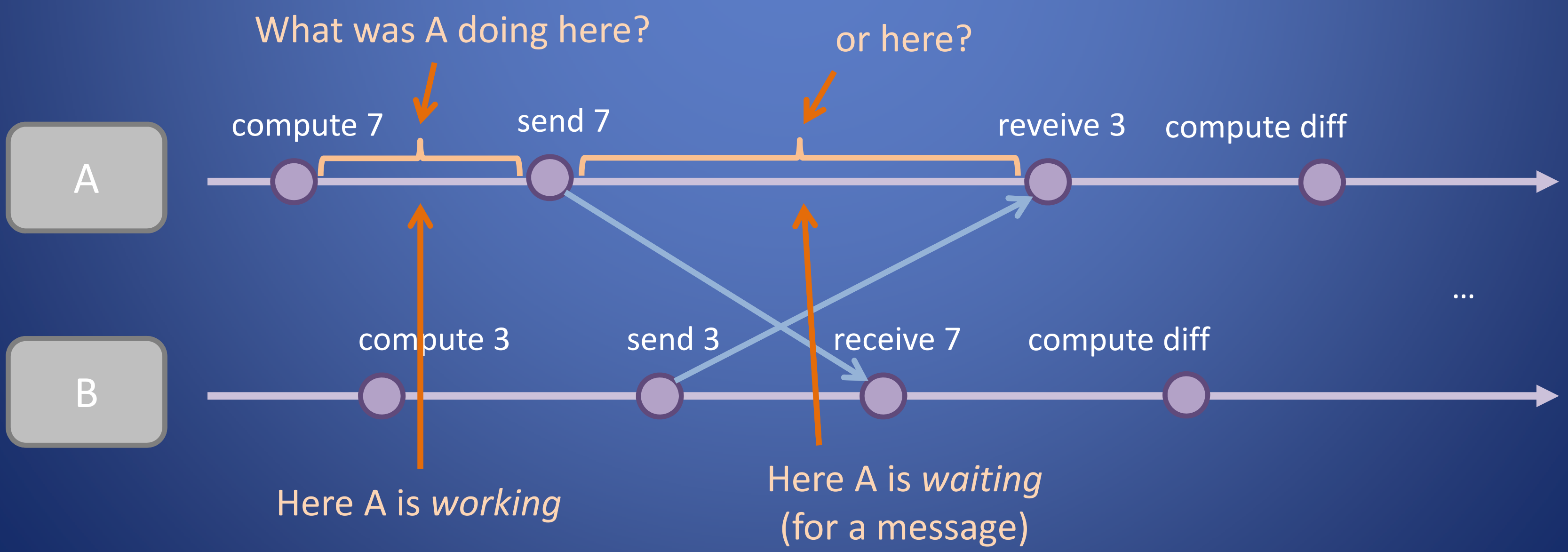


Animimation of Algorithm Instance



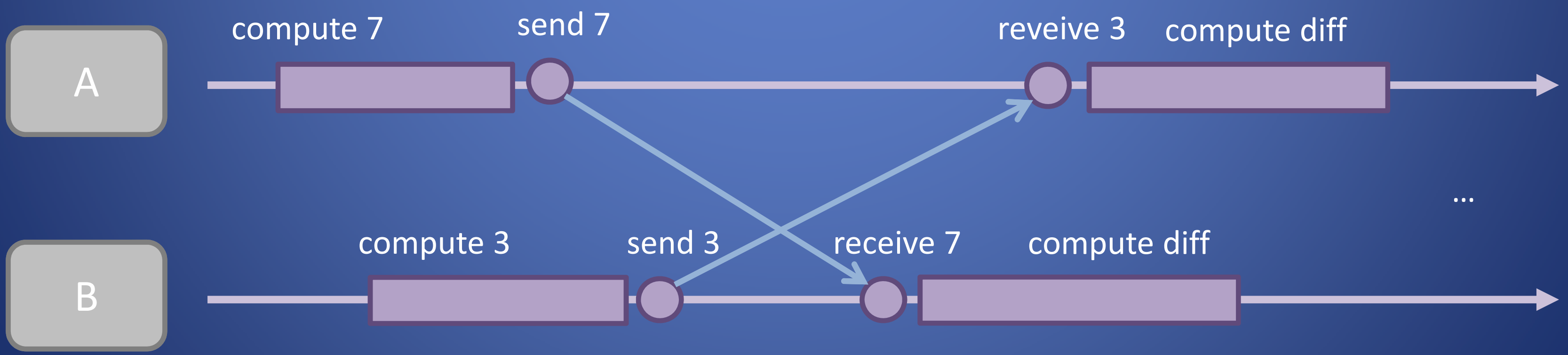
Timing Diagram of Algorithm Instance

- Show order of events (or time) at each node as a long horizontal or vertical arrow
  - After all, nodes are independent / concurrent
- Draw an arrow from each send event to its corresponding receive event



Timing Diagram of Algorithm Instance

- Show computation events as thick bars
- Absence of bar means waiting for a message
- *Questions:*
  1. How does a node's program "wait"?
  2. What happens if a message never arrives?



Answers, for our example

- Our example's **`receive()`** call *blocks*
  - If input channel is empty, node execution suspends until remote node enqueues a message
- Problem: if remote node never enqueues a message we'll wait "forever"!
  - A new, exciting way for programs to run forever (in addition to infinite loops in sequential program)
  - We'll say more about failures later

Blocking of Sends

- We could model channels as having infinite space for messages (even more perfect!)
- But we'll be more realistic and say: channels have a finite capacity.
- Hence, **send()** can also block, when channel is *full*, with no space for additional messages
  - Execution resumes once remote node dequeues a message, freeing up space in the queue

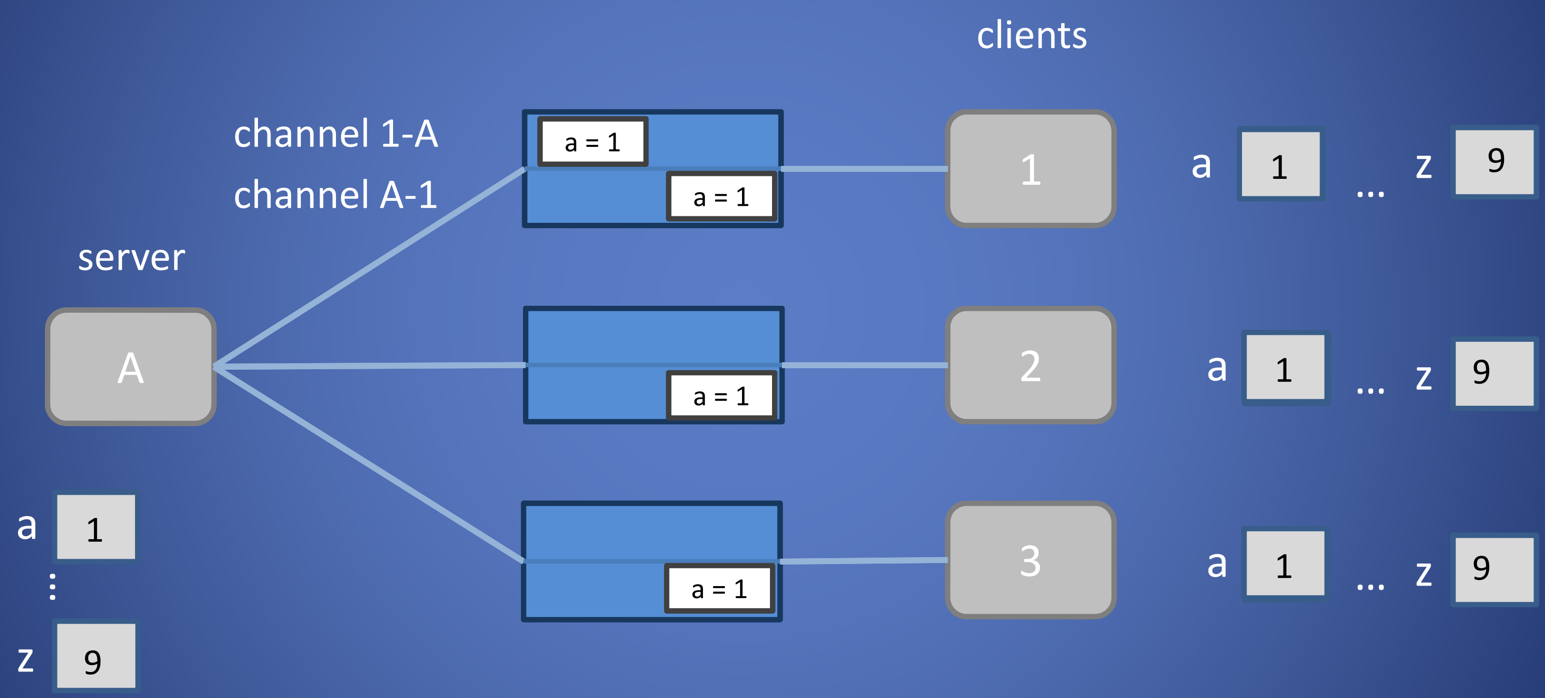
To Block, or Not to Block

- Pro: blocking is relatively simple/easy
  - Sends and receives look like computation events, program looks a lot like a sequential program
  - Terminology: the execution appears *synchronous*
    - System execution is *deterministic,* given start state
  - Waiting is *implicit:* programs don't check them but proceed as if they're always in a ready state
- Con: can limit performance and interaction styles
  - Suspending / resuming execution carries costs
  - Strict request / response messaging can be restrictive

Non-Blocking Alternative: Polling

- Add new non-blocking event: **`receive_if()`**
  - Returns a message if queue non-empty or else a "queue empty" indicator
  - Node can go do something else, when queue empty
  - New **`send_if()`** event may return "queue full"
- Program acknowledges time, is *asynchronous*
  - System is now inherently *non-deterministic*
- Permits one node to handle multiple queues
  - Poll them in turn, handle those that are ready

Example: Publish/Subscribe w Polling



Client *n* Code for Publish/Subscribe

```
do forever:
   msg = receive_if(A-n)
   if msg ≠ "queue empty":
      var, val = unpack contents of msg
      update variable "val" with value "val"
   … compute something for a while …
   for each variable var I want to set to value val
      msg = pack "var" and "val " into a message
      if send_if(n-A, msg) = "queue full":
         exit
```

Alternative: wait a little while, then try again

Server Code for Publish/Subscribe

```
sndChannels = {A-1, A-2, A-3}
do forever:
   for rch in {1-A, 2-A, 3-A}:
      msg = receive_if(rch)
      if msg ≠ "queue empty":
         var, val = unpack contents of msg
         update my variable "val" with value "val"
         for sch in sndChannels:
            if send_if(sch, msg) = "queue full":
               remove sch from sndChannels
```

Real Network Channels Fail!

- We can model such *unreliable* channels:
  - Asked to enqueue, channel might:
    - Do nothing at all (*drop* messages)
      - Note: same as **`send_if()`** with a full channel
    - Append a different msg (*corrupt* messages)
  - Asked to dequeue, channel might:
    - Remove and deliver a different msg (*reorder* messages)
    - Deliver a msg but not remove it (*duplicate* messages)
- Example: Internet's User Datagram Protocol (UDP)
  - Msg drops, reorders, duplicates

Queue Model of UDP/IP

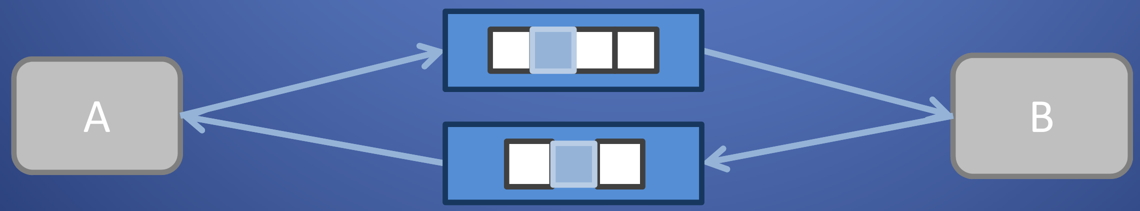
- Each *network interface* of an Internet device is identified by a globally unique *IP Address*
  - A 32-bit integer, e.g. 82D0F047 hexadecimal
  - Written as dot-separated decimals, from most to least significant byte, e.g. 130.208.240.71
- A UDP "channel" comprises an IP Address and a UDP *Port*: a 16-bit integer
  - Ports below 1024 are allotted by convention to "well known services", such as DNS.
    My main DNS server is at 46.22.96.35 : 53

Sending / Receiving UDP Messages

- UDP is *connectionless*: you send a message to a channel anytime (via OS's APIs, e.g. socket)
  - But you have no idea if it gets delivered or not
  - Can be up to ~64KB in size, but prefer < 1500 bytes, or a few KB at most
- To receive UDP: *bind* as a listener of some port *P* (via OS's API, e.g. socket)
  - You will receive (a subset of the) UDP messages sent to channel: your-IP-Address : *P*

Example: Reliable Communication

- Want to exchange an ordered sequence of messages over an unreliable channel that drops, duplicates and reorders messages
  - This is what TCP provides, on top of the unreliable Internet Protocol (IP) packet delivery service
  - UDP is a very thin layer on top of IP



Reliable Messaging: Sender Protocol

What's a good value for "little while"?

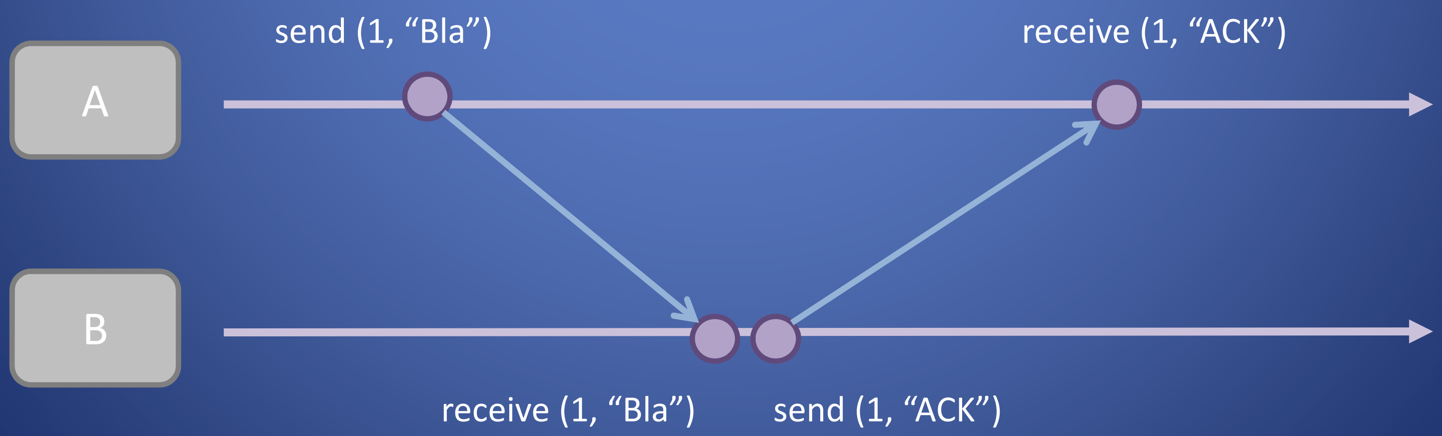
```
global numSent = 0
// channel now represents both send and recv queues
function reliable_send(msg, channel):
   numSent = numSent + 1
   do forever:
      send_if(channel, (numSent, msg))
      wait for a little while
      reply = receive_if(channel)
      if reply ≠ "queue empty":
         numReceived, msg = unpack reply
         if msg = "ACK" and numReceived = numSent:
            return
```

Reliable Messaging: Receiver Protocol

```
global numReceived = 0
// channel now represents both send and recv queues
function reliable_receive(channel):
    do forever:
        packet = receive_if(channel)
        if packet ≠ “queue empty”:
            packetNum, msg = unpack packet
            if packetNum = numReceived + 1
                numReceived = numReceived + 1
                send_if(channel, (“ACK”, numReceived))
                return msg
        send_if(channel, (“ACK”, numReceived))
        wait a little while
```

Let’s Check our Protocol

- The channel is our *adversary*: it misbehaves and tries to confuse us. Try protocol with:
  - Dropped, re-ordered, duplicate messages
  - Dropped, re-ordered, duplicate ACKs
- Below is the failure-free, happy case:



Take-home Points

- *Designing robust network protocols is difficult*
  - Have to anticipate and handle every type of failure that can occur, at any stage in the protocol
  - The Message Queue/Event model can help a lot
- Use existing building blocks whenever possible
- For example: UDP is rarely beneficial. Better to use a reliable transport, like TCP
  - You'll end up re-implementing TCP anyway
  - Possible exception: fast-paced networked games

TCP vs. Our Toy Protocol

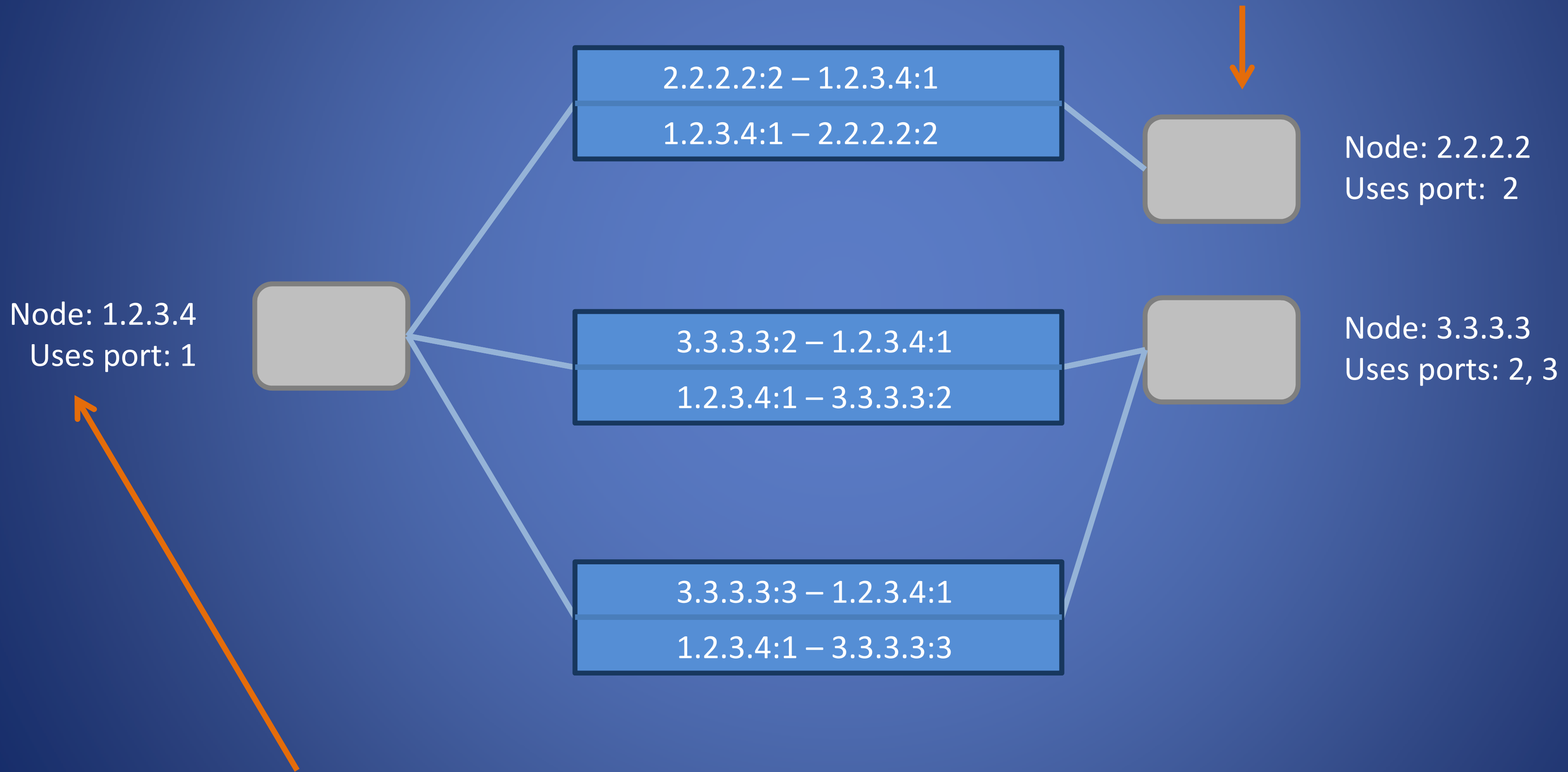
- Transmits byte sequences, not discrete messages
  - You send a byte buffer, TCP chops it up into *segments* (packets) any way it pleases, ACKs byte seq positions.
  - You must provide message *framing*, e.g. prepend the length of your messages to their data
- Buffers sent and received data and has multiple segments "in flight" on network at the same time
  - Message-by-message "ping-pong" would be way to slow
- Performs *flow-control* and *congestion avoidance*
  - Adjusts transmission rate to current network bandwidth and shares bandwidth fairly with other connections

Queue Model of TCP/IP

- TCP is *connection-oriented*: you establish a *connection* with a remote node before exchanging messages with it
  - To agree on initial sequence numbers, etc.
- We can model this as creating a new channel
  - We thought of UDP channels as pre-existing
- A TCP channel is globally/uniquely identified by *two* IP Address:Port pairs
  - The IP Addresses of the two nodes involved

Queue Model of TCP/IP, Continued

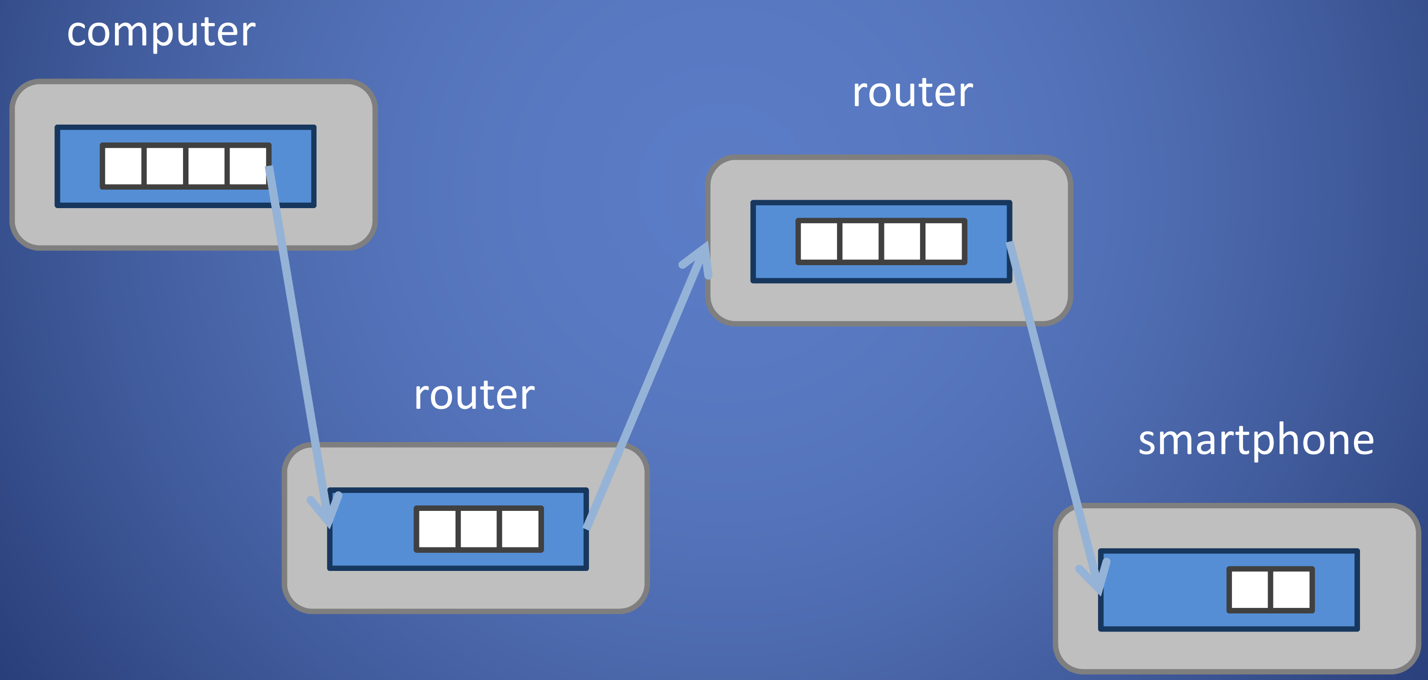
These nodes are playing the roles of clients connecting to server. They choose their ports at will.



This node is playing the role of a server, accepting connections at a “well-known” port
(e.g. port 80 for http, the World Wide Web protocol)

Queues Are Real!

- Networking hardware/software full of queues

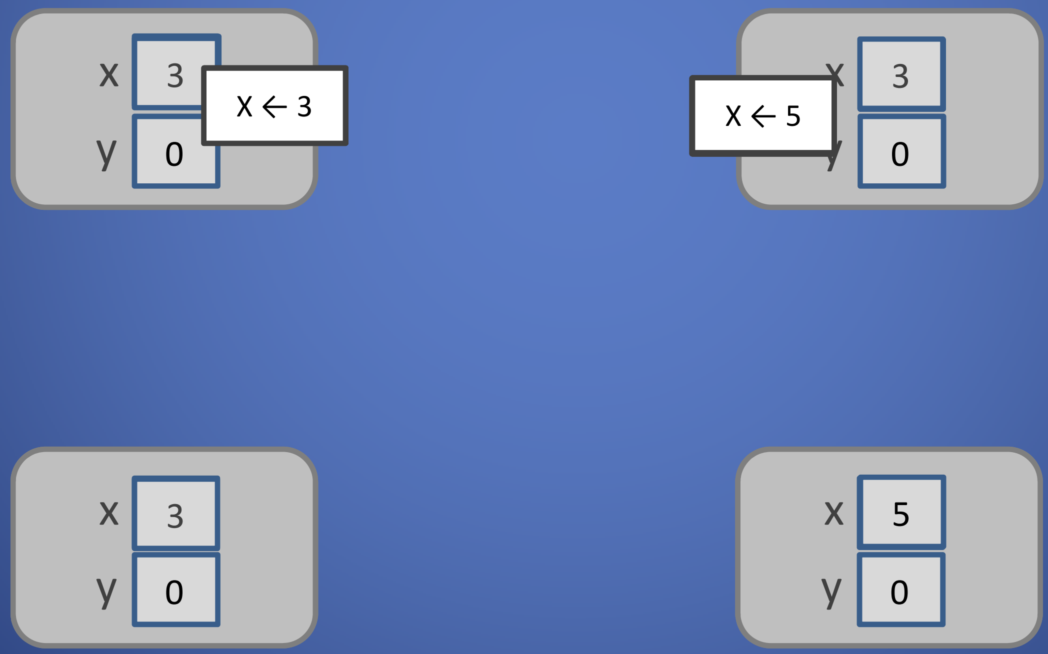


Modeling Multi-User Games

- Multiple nodes hold a copy of some state
- We want them to behave as if there was a single *shared* instance of the state
  - They can't really, can only exchange messages
- Nodes that (propose to) mutate state must notify other nodes, which update their copies
- Problem: nodes can *diverge*: breaking illusion
  - Can mutate differently or in different order

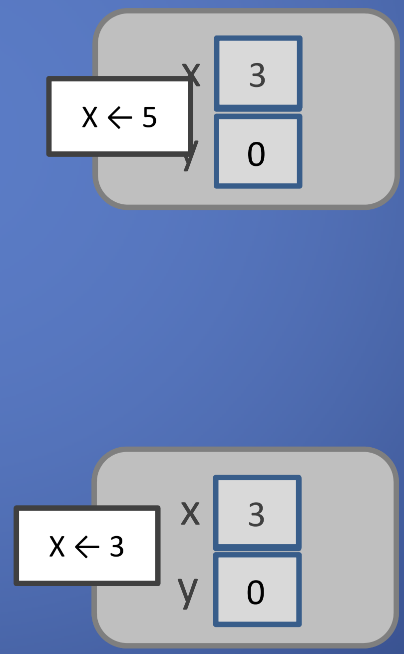
Replicated System Problems

- First-Order problem: conflicting updates



Near-Universal Solution: Master/Slave

- One node is the *master* for updates, the other *slave* nodes forward their updates to master



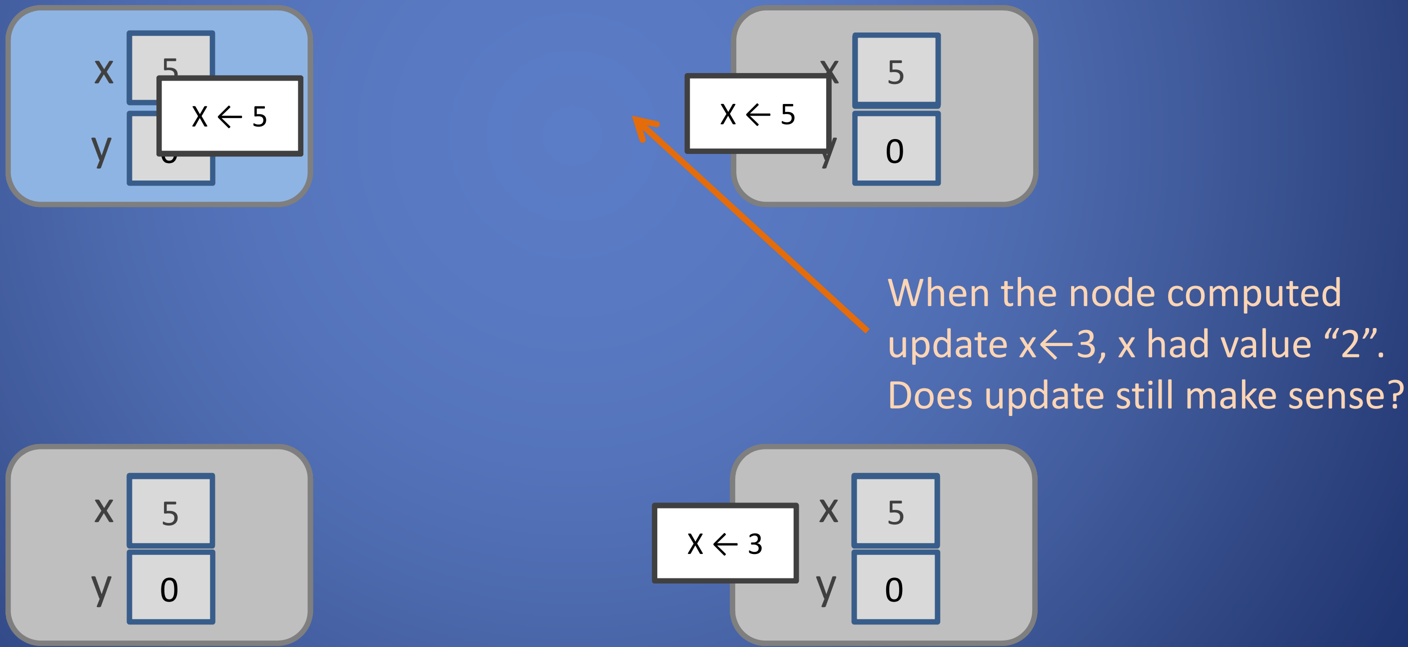
In essence: we ensure everyone's receive queue looks the same as the master's queue

So far, so good but …

- What we've shown is basically a *distributed cache*, where slaves are *eventually consistent*
  - Master is *authoritative*. It is in a position to authenticate, modify or reject changes
  - MMOs usually have a permanent, trusted master (operated by game corp) since end-user cheat!
- Problem: a slave's decision to mutate may have been based on *stale* (old, obsolete) data
  - For example: shot a dude who had moved away

Inconsistent Execution / Race Condition

- The state is shared but the *simulation is not*



Solution 1: DB-Style Distributed Locking

1. Slave sends master a request to *lock* the set of variables it wants to read and/or update
2. The master acknowledges the request, if no other node has any of the variables locked
   – Otherwise: rejects or delays the lock request
3. Slave then executes event and sends update
   – No inconsistency, other's can't modify the vars
4. Master updates and unlocks

Problems with Locking

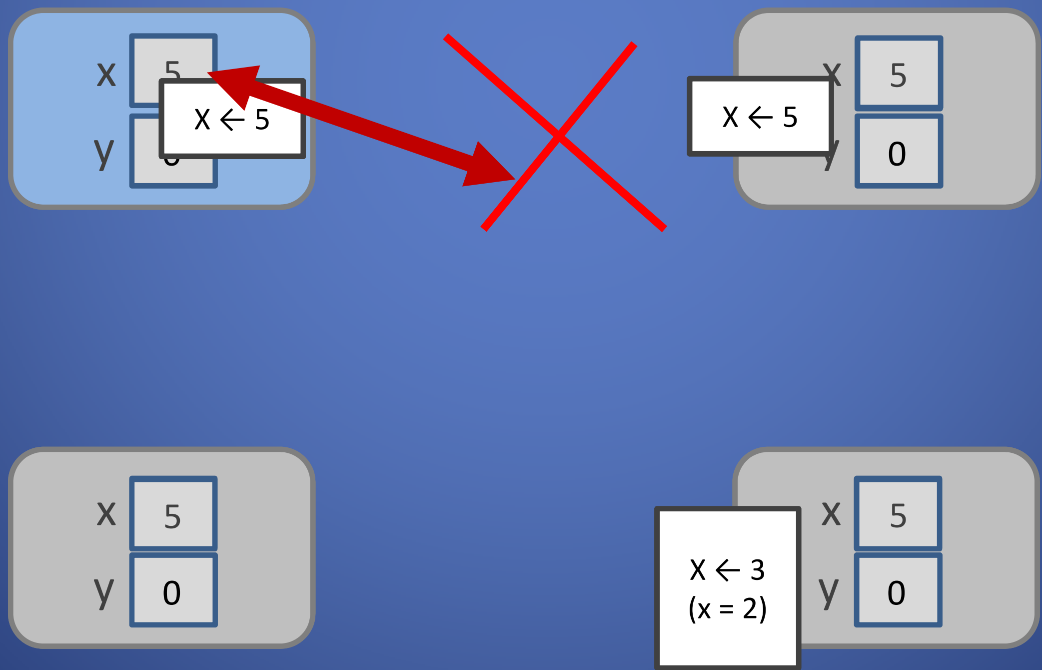
- Low performance: slaves spend at least a *message round-trip* waiting, for each update
  - *This alone rules out locking for most games*
- Fault-tolerance: if slave crashes or loses connection, variables left in locked state
- Deadlocks: lock requests can form circular wait-for dependencies
  - Not a biggie, master can detect such cycles and break them by rejecting one of the lock requests

Solution 2: Optimistic Concurrency Control

- Give master enough information to be able to reject updates based on (possibly) stale data
  - Slave sends with updates the *read set* of variables read by event's execution, as well as their values
  - Master checks if all of an update's read-variables still have these value. If not, rejects update
    - Alternative: master tracks which updates each slave has received and rejects updates if any read-set value has changed (disregarding values)

Optimistic Conc Ctrl in Action

- Server verifies updates were made assuming correct variable values

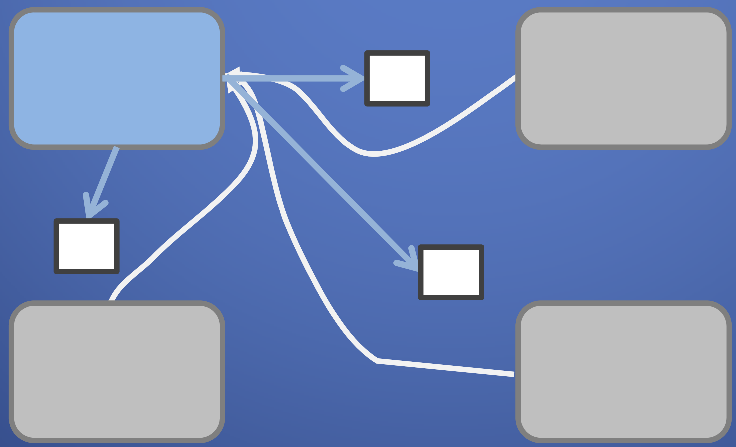


Optimistic Concurrency Control Pro & Con

- Pro: when there are no conflicts, there is no waiting and no additional delay
- Con: read-sets can be large, eat network bandwidth
- Con: high *contention* (many conflicts) may cause *livelock* : some slave keeps losing out
  - For example: a slave with high network latency
  - Can be hard to ensure *fairness* for all nodes

Solution 3: Share Execution, not Updates

- Instead of sending state mutations, slaves send user input (mouse/keyboard) to master
- Master executes total simulation and distributes resulting state updates to slaves



Shared Execution Pro & Con

- Pro: works well, this is essentially how most quick-paced games do it (FPSes, e.g.)
  - Games no longer treated as a database problem
- Con: centralized master limits scalability
- Con: large delay from mouse/keyboard action to effect on screen (e.g. turning head)
  - On the order of a network round-trip, 10s of ms
  - Makes players sick / drives them crazy

Solutions to Shared Execution Delay

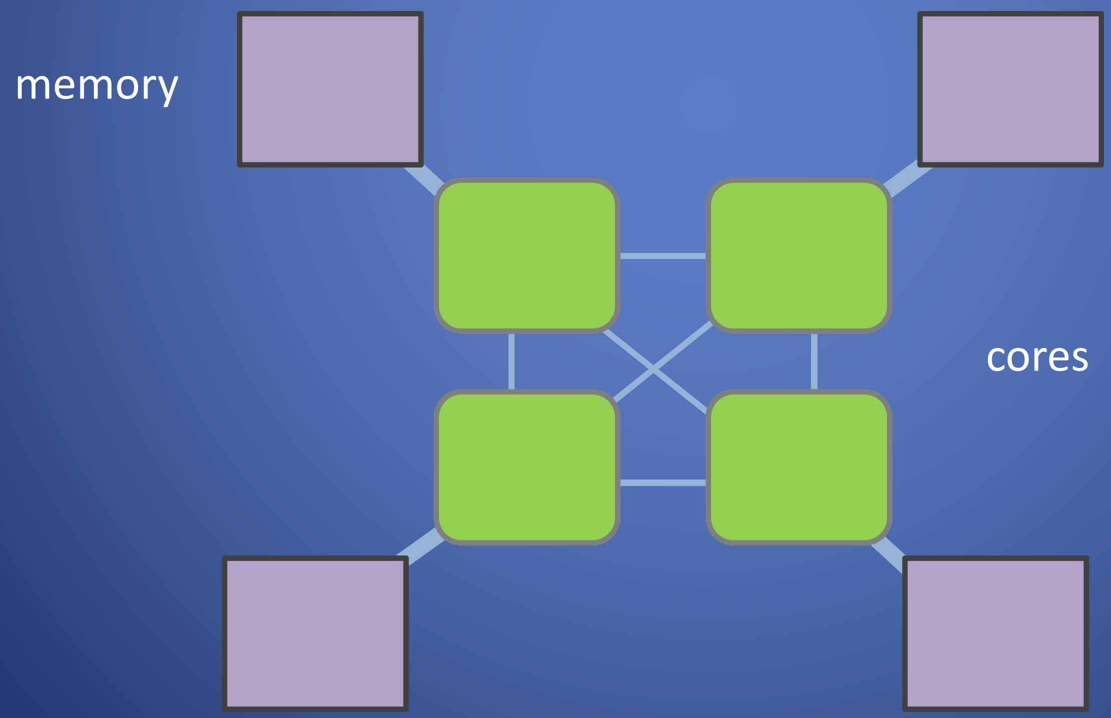
- *Prediction:* slaves also execute game logic, assuming immediate effect of user's input
  - Predict how player's character moves, predict how other user's characters will move.
- When master sends actual / authoritative updates, slaves must *reconcile* their local version using updates, converge to master
  - Shift characters towards correct position, e.g.

This is not a fully solved problem

- FPS engines (Quake, Unreal …) have finely hand-tuned, fairly ad-hoc solutions
  – Separate predictions for character running, jumping, gun shots, flying grenades …
  – Heavily optimized/compressed encoding of update packets, to conserve bandwidth
- Can be solved generally through determinism
  – Slaves *roll back* their state to time of new server update and the *replay* all events back to now
  – As you figure it out, use the Queue, Luke!

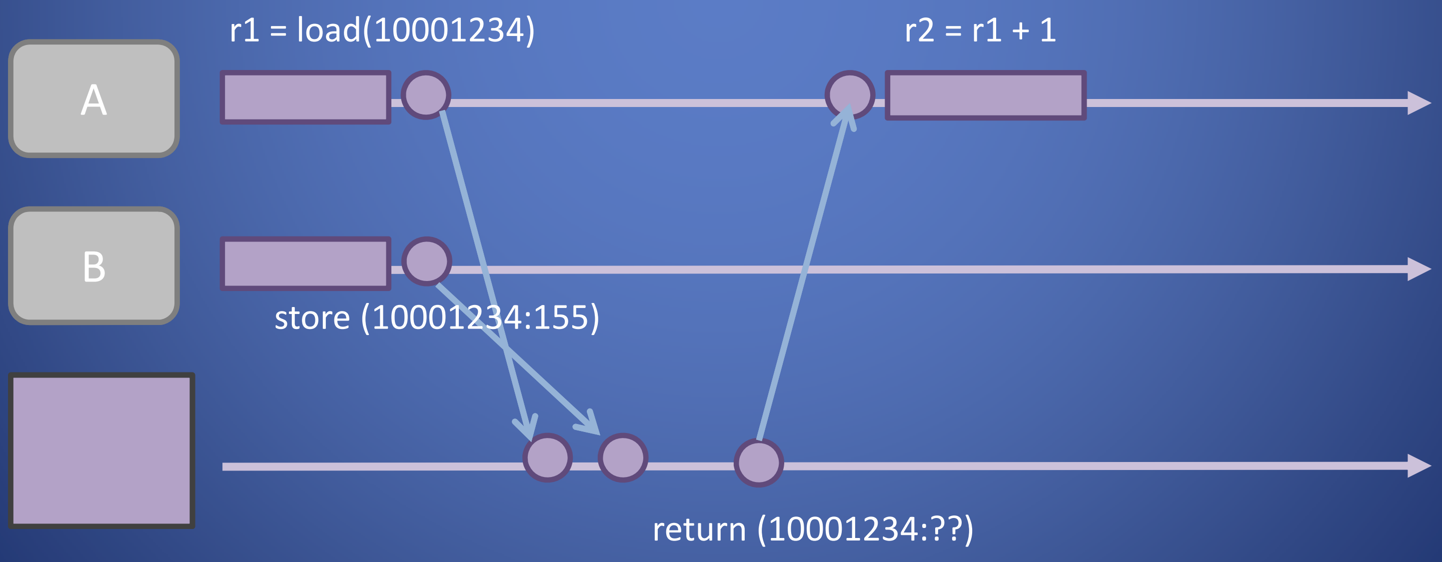
“Distributed” Systems Everywhere!

- Multi-core machines (with NUMA)
  - Fast, failure-free networks (memory, PCI Express)



“Distributed” Systems Everywhere!

- Shared-memory Threads: can model as nodes
  - Memory accesses are message passing
    - Implemented by memory controller hardware



Summary

- Modeling distributed systems as nodes exchanging messages via queues is very useful
  - This is how academics do it, for their proofs!
- Shared state is the canonical hard problem for distributed systems
  - We've seen the top of the iceberg today. Add partial failures, partial subscriptions, partitioned servers, dynamic migration …
- MMOs are special, but not all that special
  - Yet to successfully apply knowledge from DB/Distr