CS162
Operating Systems and Systems Programming
Lecture 22

Distributed Decision Making (Finished), TCP/IP Networking, RPC

April 21st, 2020

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. *When slides are obtained from other sources, a a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

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

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two machines agree to do something, or not do it, atomically
- Two-Phase Commit protocol:
  - 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
- Persistent stable log on each machine:
  - Help nodes remember what they have said that they would do
    - » If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
    - » Log can be used to complete this process such that all machines either commit or don't commit

Two-Phase Commit: Setup

- One machine *(coordinator)* initiates the protocol
- It asks *every* machine to vote on transaction

- Two possible votes:
  - Commit
  - Abort

- Commit transaction only if unanimous approval

Two-Phase Commit: Preparing

Agree to Commit

- Machine has guaranteed that it will accept transaction
- Must be recorded in log so machine will remember this decision if it fails and restarts

Agree to Abort

- Machine has guaranteed that it will never accept this transaction
- Must be recorded in log so machine will remember this decision if it fails and restarts

Two-Phase Commit: Finishing

Commit Transaction

- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction

- Coordinator learns *at least on machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters

Two-Phase Commit: Finishing

Commit Transaction

- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction

- Coordinator learns *at least on machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters

Because no machine can take back its decision, exactly one of these will happen

Detailed Algorithm

Coordinator Algorithm

Coordinator sends **VOTE-REQ** to all workers

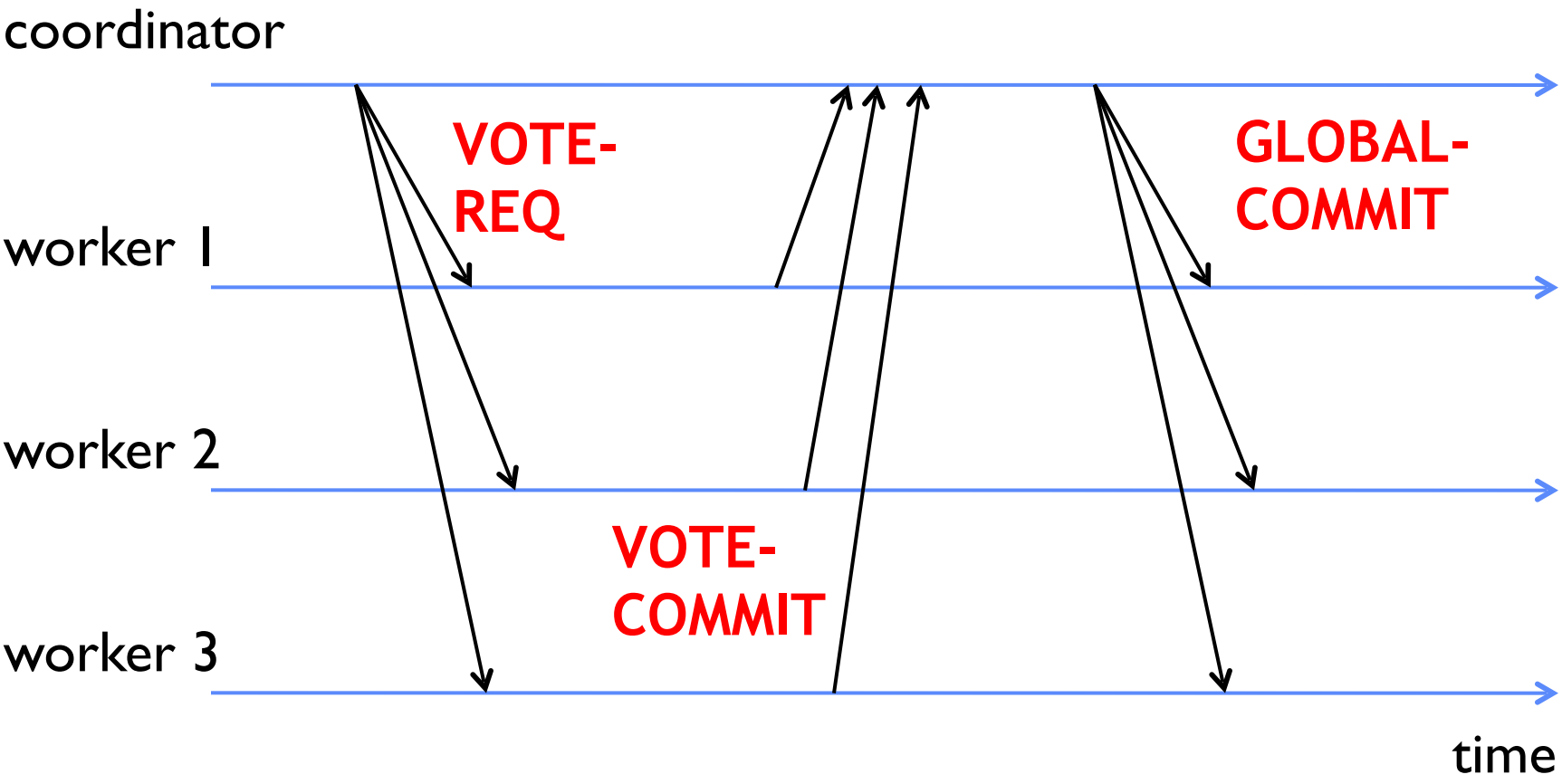
- If receive **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If don't receive **VOTE-COMMIT** from all N workers, send **GLOBAL-ABORT** to all workers

Worker Algorithm

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

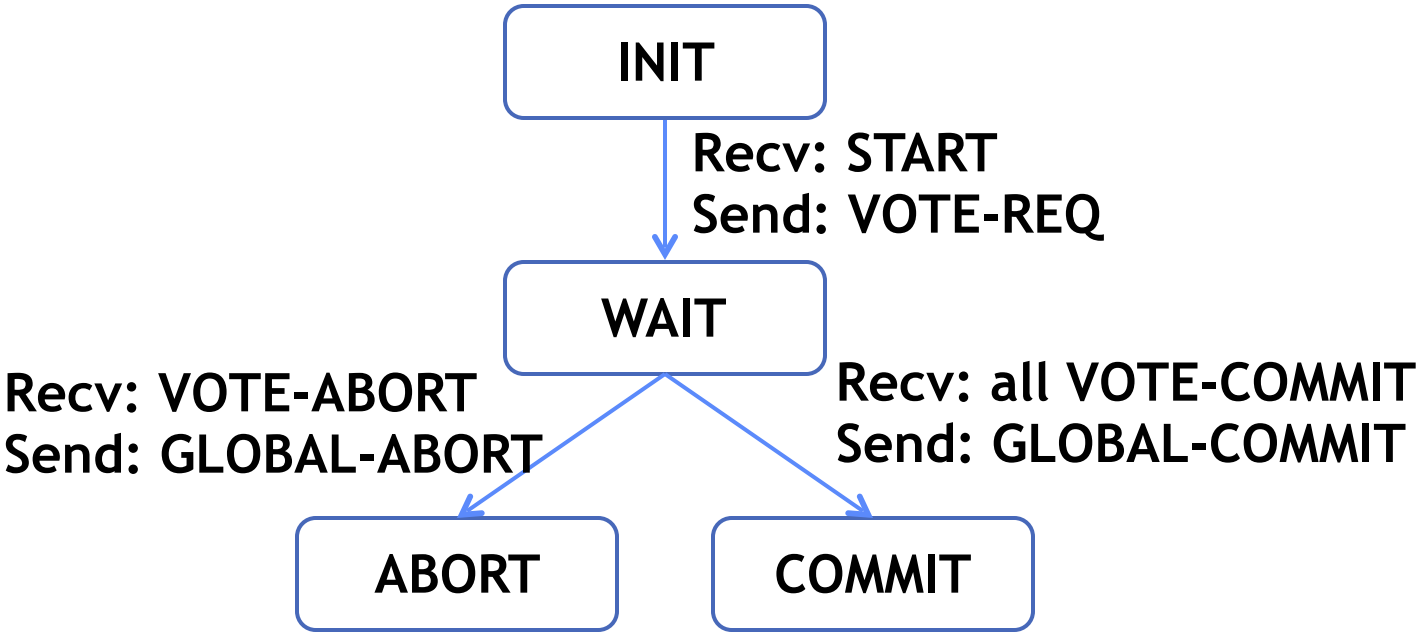
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

Failure Free Example Execution

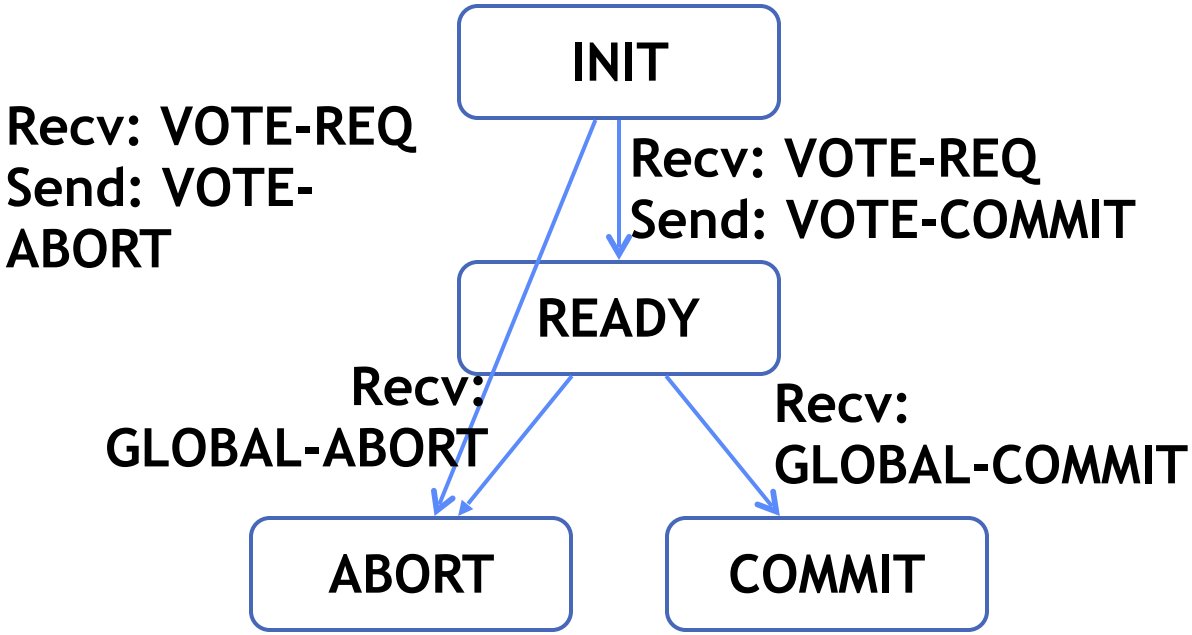


State Machine of Coordinator

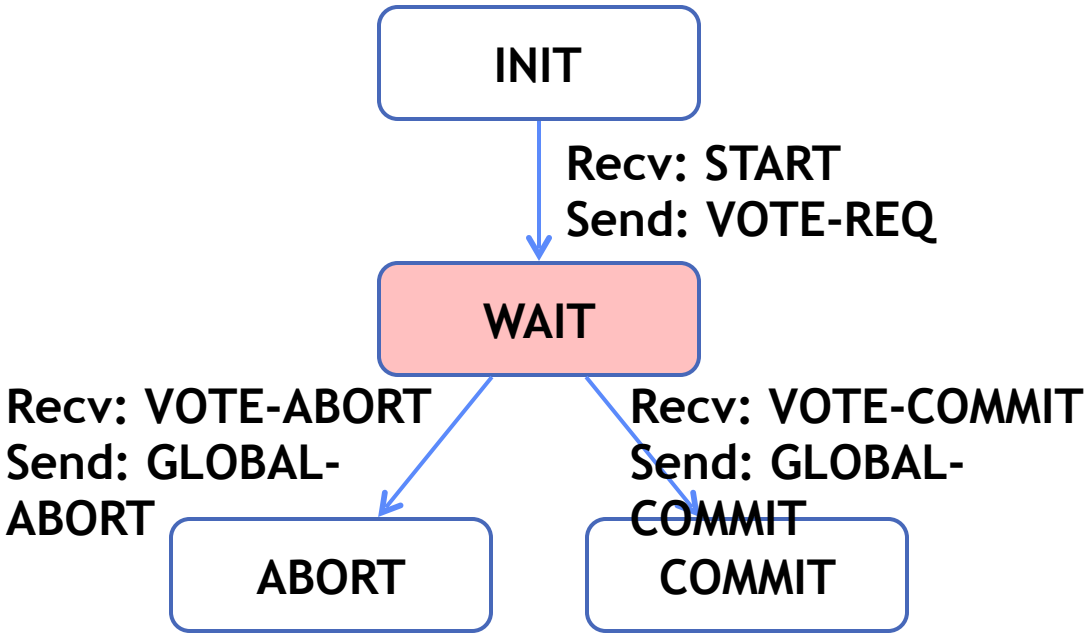
- Coordinator implements simple state machine:



State Machine of Workers

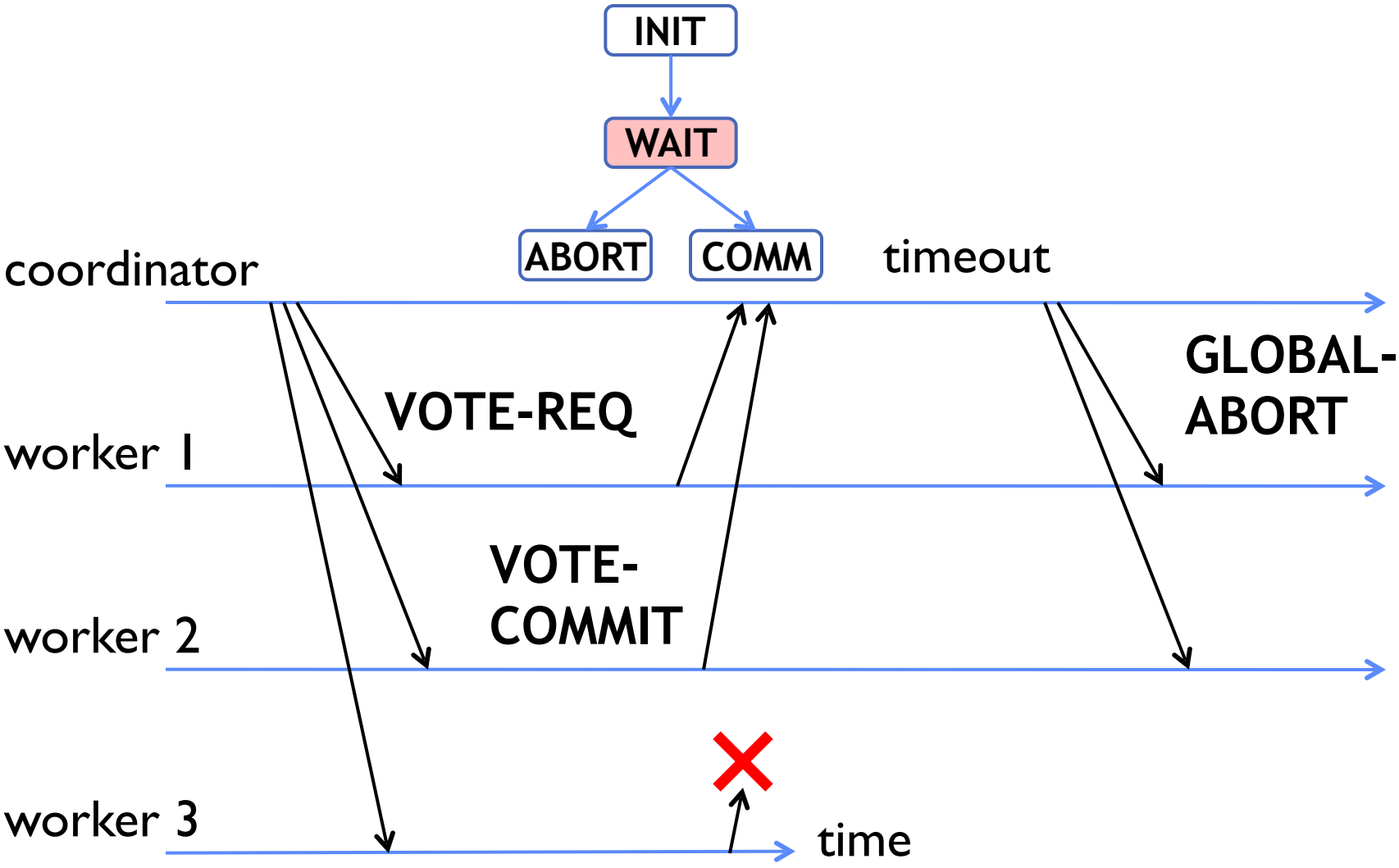


Dealing with Worker Failures

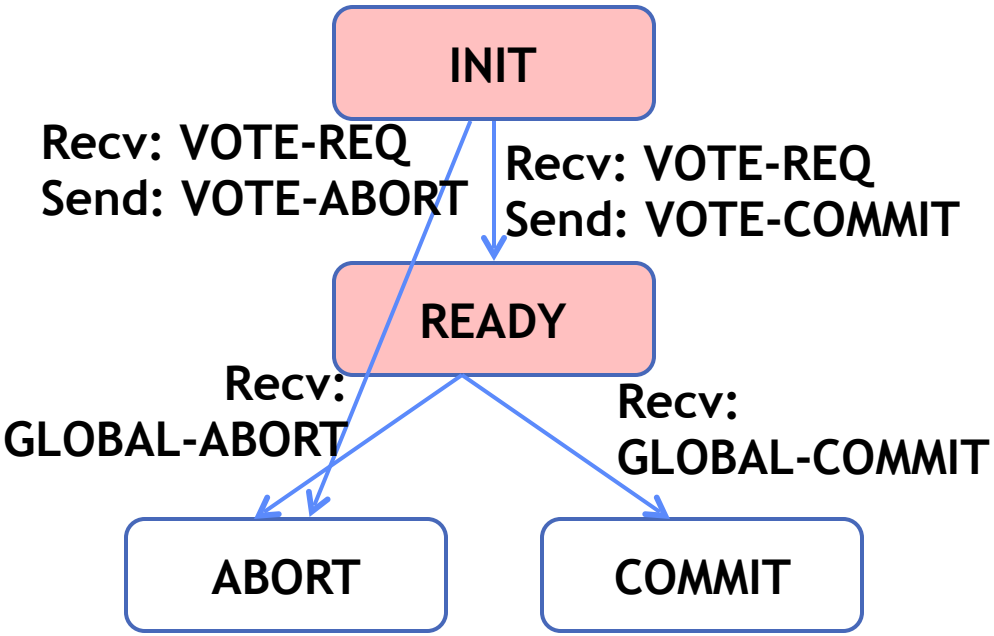


- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in "**WAIT**" state
- In **WAIT**, if doesn't receive N votes, it times out and sends **GLOBAL-ABORT**

Example of Worker Failure

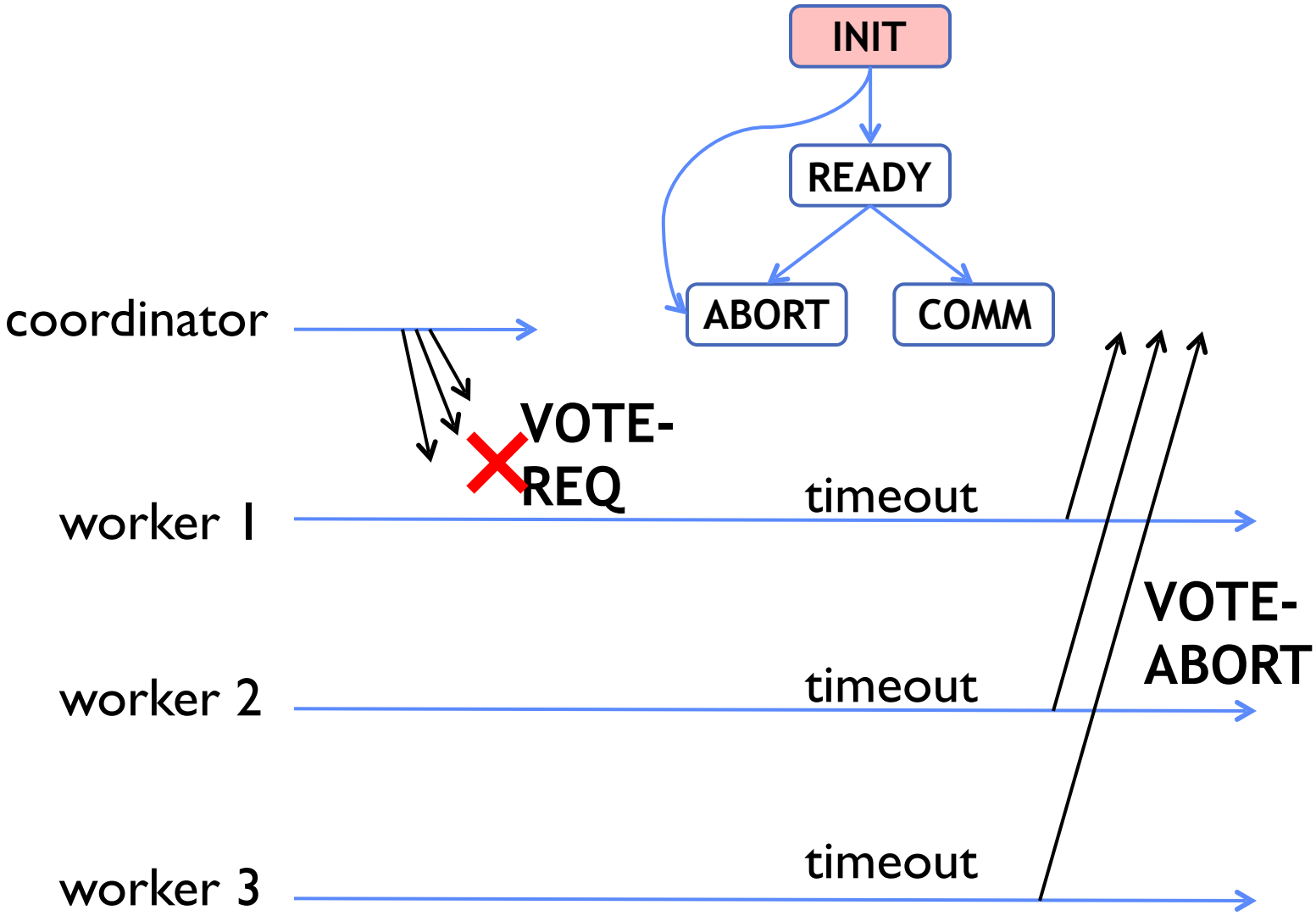


Dealing with Coordinator Failure

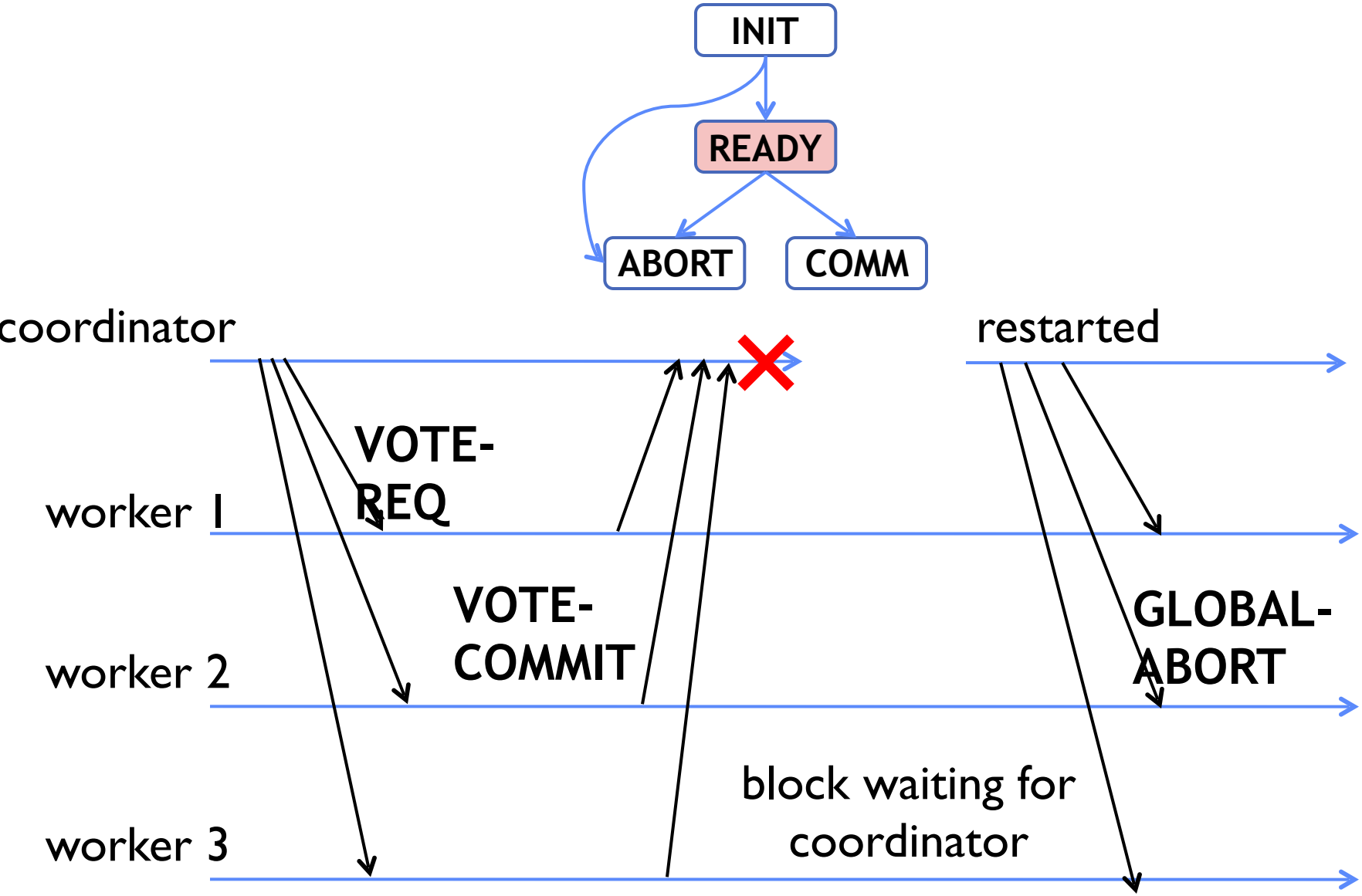


- Worker waits for **VOTE-REQ** in **INIT**
  - Worker can time out and abort (coordinator handles it)
- Worker waits for **GLOBAL-*** message in **READY**
  - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send **GLOBAL_*** message

Example of Coordinator Failure #1



Example of Coordinator Failure #2

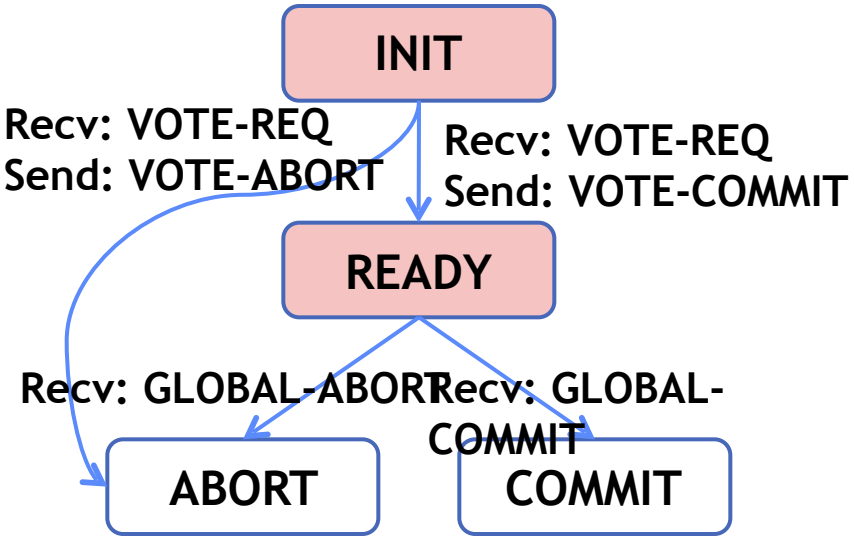


Durability

- All nodes use stable storage to store current state
  - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
  - E.g.: SSD, NVRAM
- Upon recovery, nodes 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
  - After decision made, result recorded in multiple places
- Why is 2PC not subject to the General's paradox?
  - Because 2PC is about *all nodes eventually coming to the same decision – not necessarily at the same time!*
  - Allowing us to reboot and continue allows time for collecting and collating decisions

Distributed Decision Making Discussion (2/2)

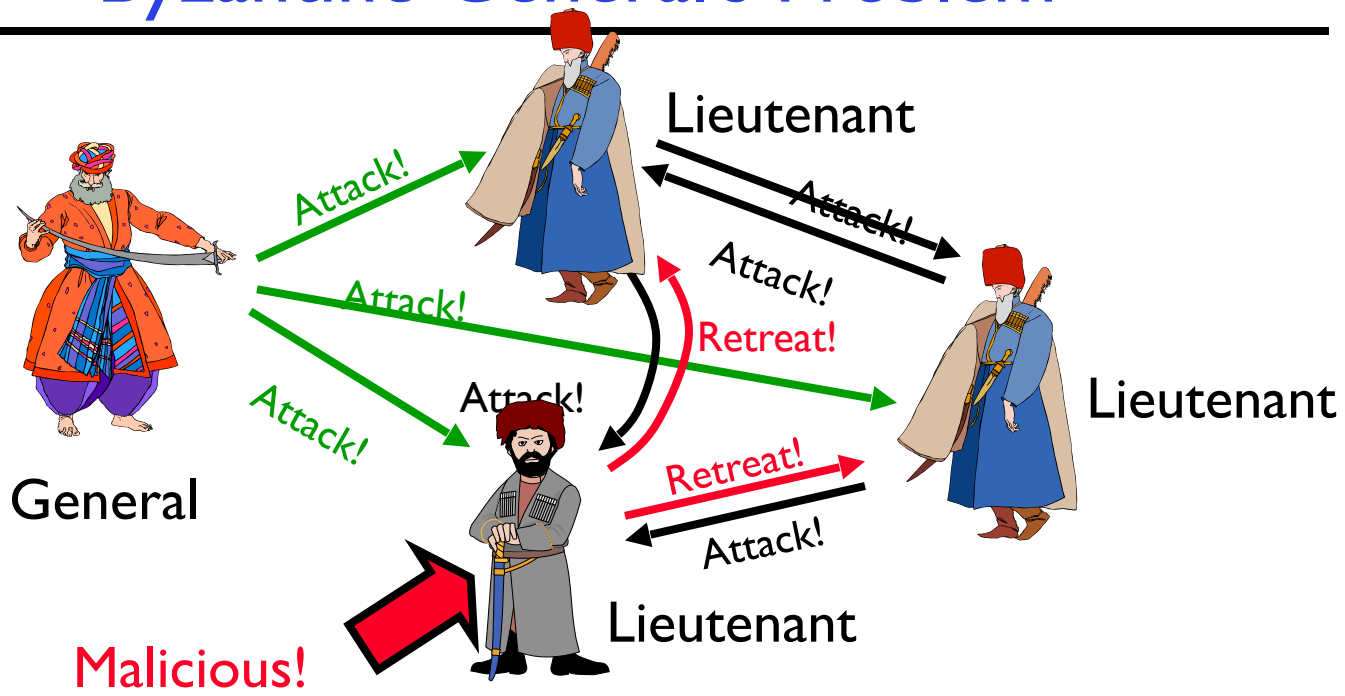
- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - » Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes
    - » Site A crashes
    - » Site B wakes up, check its log, and realizes 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

Alternatives to 2PC

- Three-Phase Commit: One more phase, allows nodes to fail or block and still make progress.
- PAXOS: An alternative used by Google and others that does not have 2PC blocking problem
  - Develop by Leslie Lamport (Turing Award Winner)
  - No fixed leader, can choose new leader on fly, deal with failure
  - Some think this is extremely complex!
- RAFT: PAXOS alternative from John Osterhout (Stanford)
  - Simpler to describe complete protocol

- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

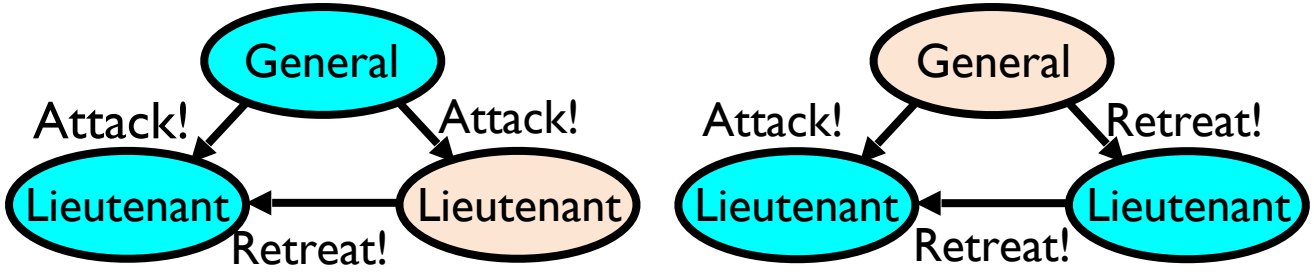
Byzantine General's Problem



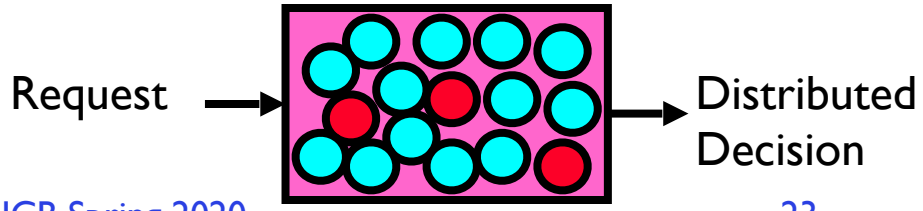
- Byazantine General's Problem (n players):
  - One General and n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that the following Integrity Constraints apply:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Byzantine General's Problem (con't)

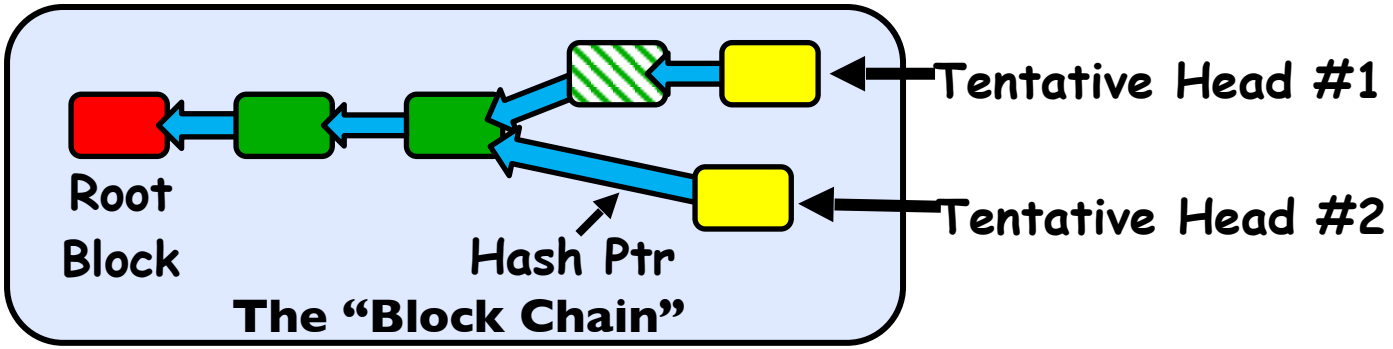
- Impossibility Results:
  - Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things



  - With f faults, need n > 3f to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity O(n2)
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them (< n/3) are malicious

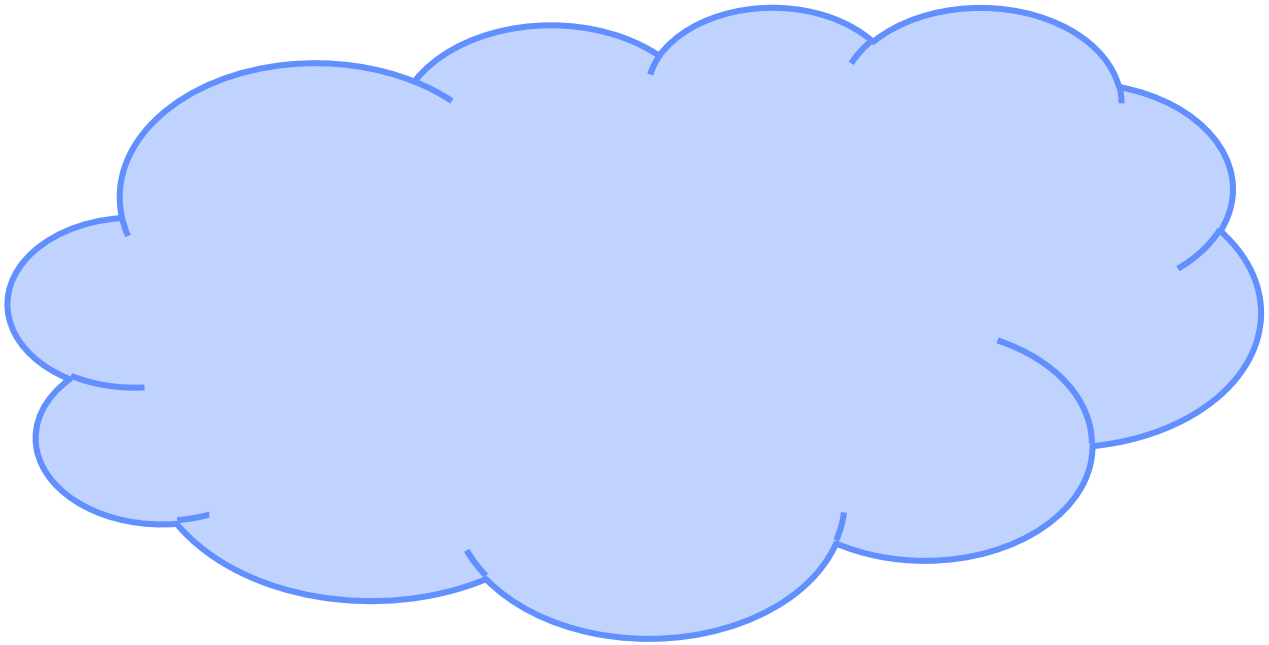


Is a BlockChain a Distributed Decision Making Algorithm?

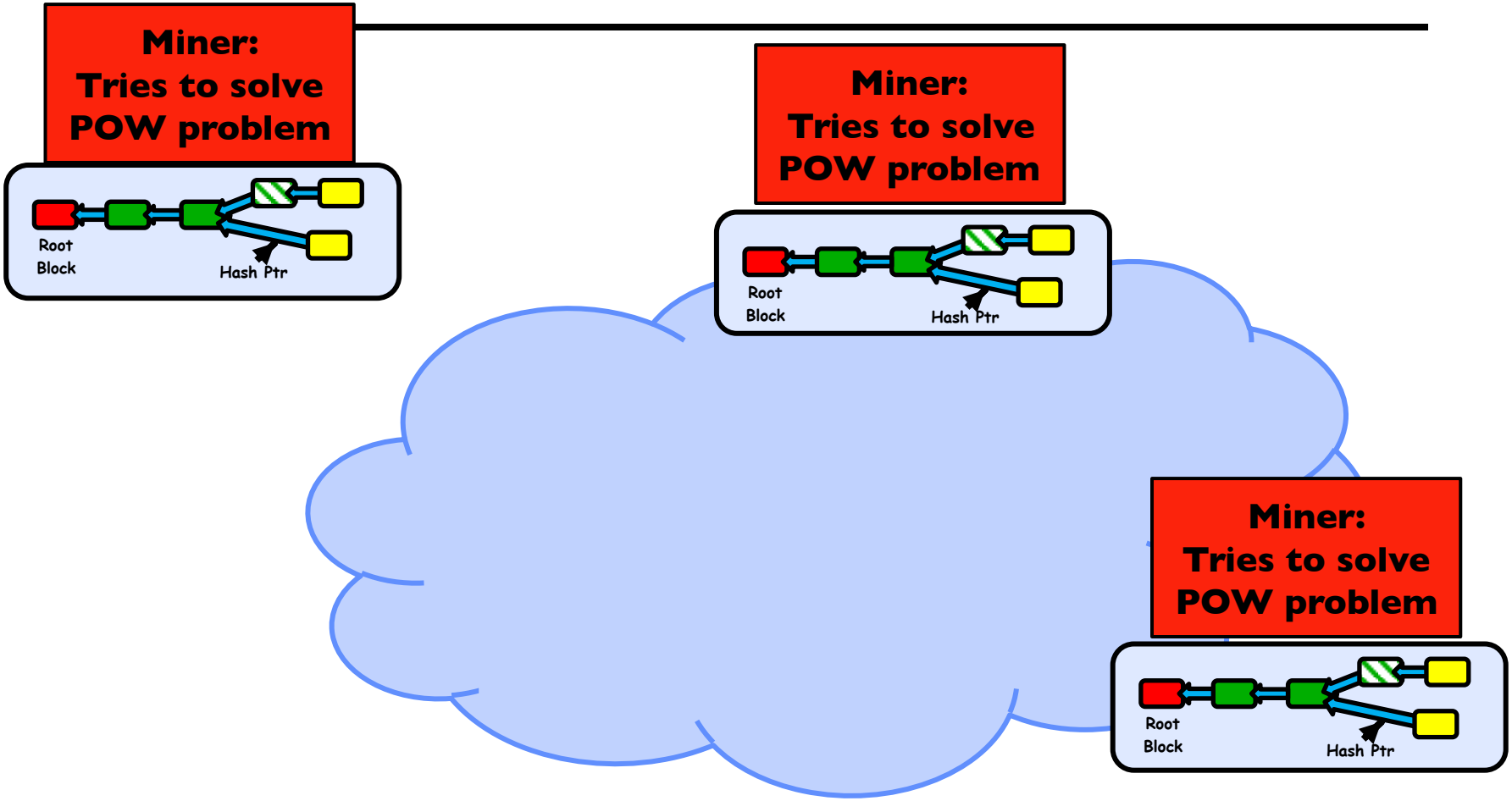


- BlockChain: a chain of blocks connected by hashes to root block
  - The Hash Pointers are unforgeable (assumption)
  - The Chain has no branches except perhaps for heads
  - Blocks are considered "authentic" part of chain when they have authenticity info in them
- How is the head chosen?
  - Some consensus algorithm
  - In many BlockChain algorithms (e.g. BitCoin, Ethereum), the head is chosen by solving hard problem
    - » This is the job of "miners" who try to find "nonce" info that makes hash over block have specified number of zero bits in it
    - » The result is a "Proof of Work" (POW)
    - » Selected blocks above (green) have POW in them and can be included in chains
  - Longest chain wins

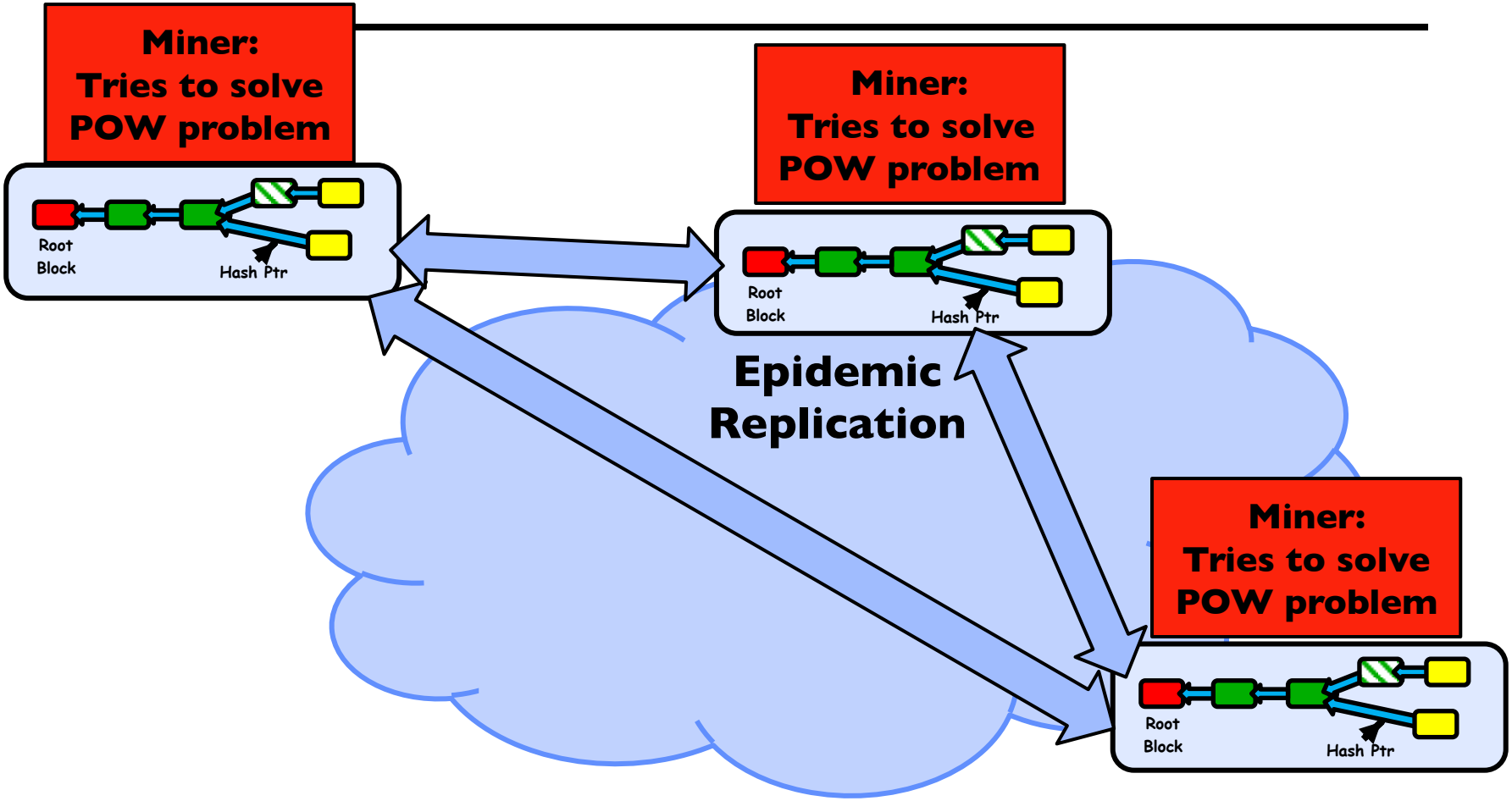
Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



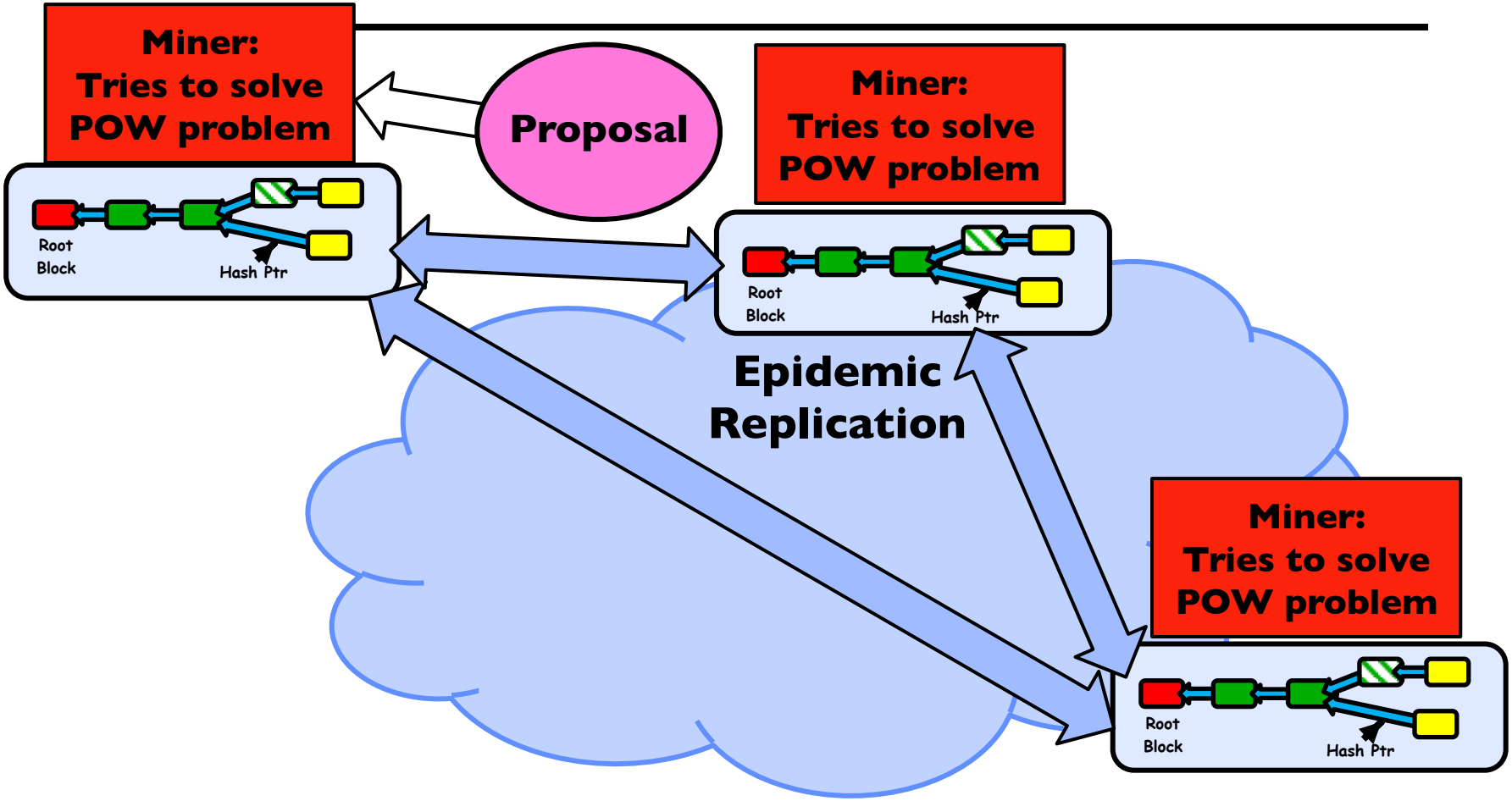
Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



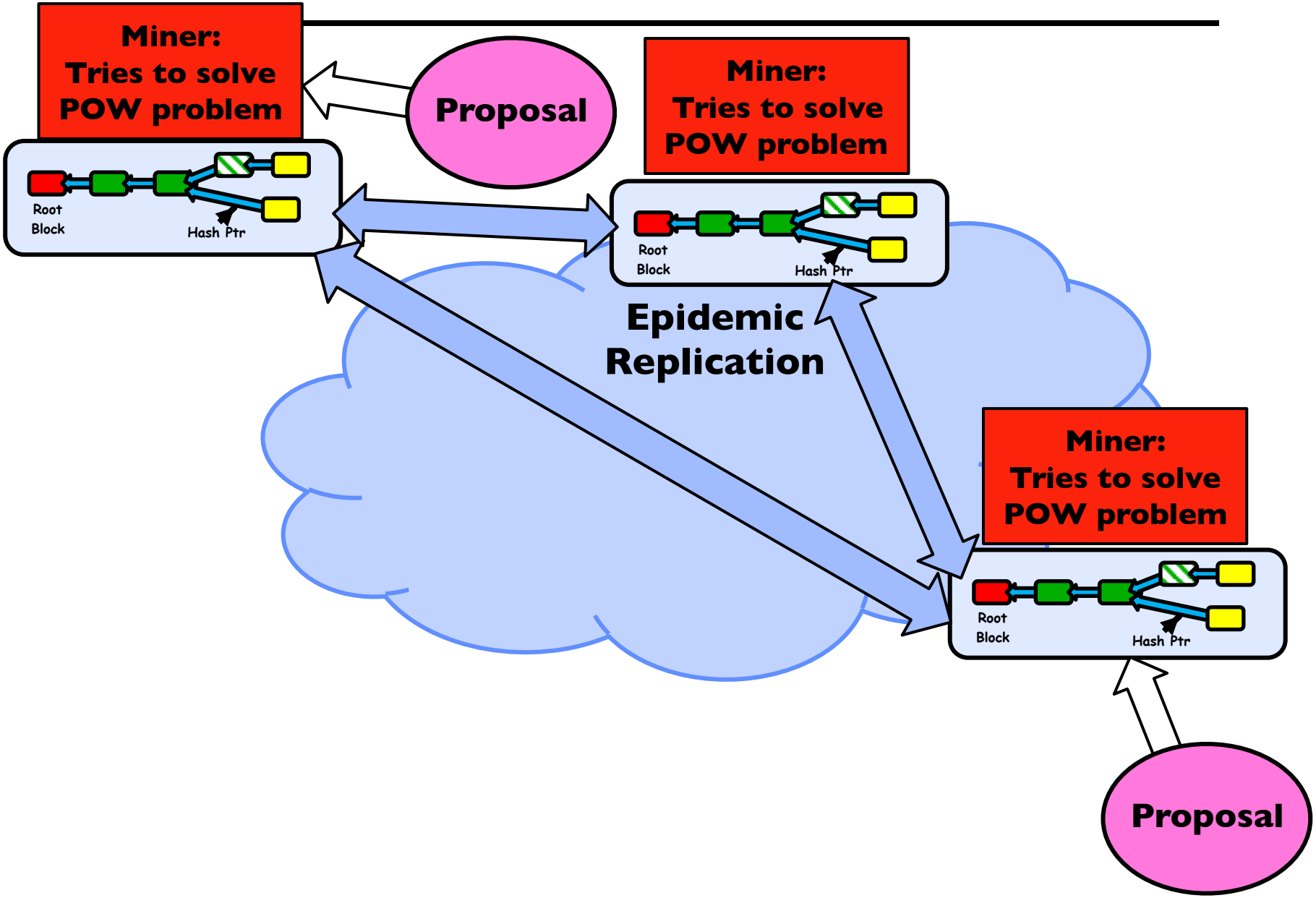
Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



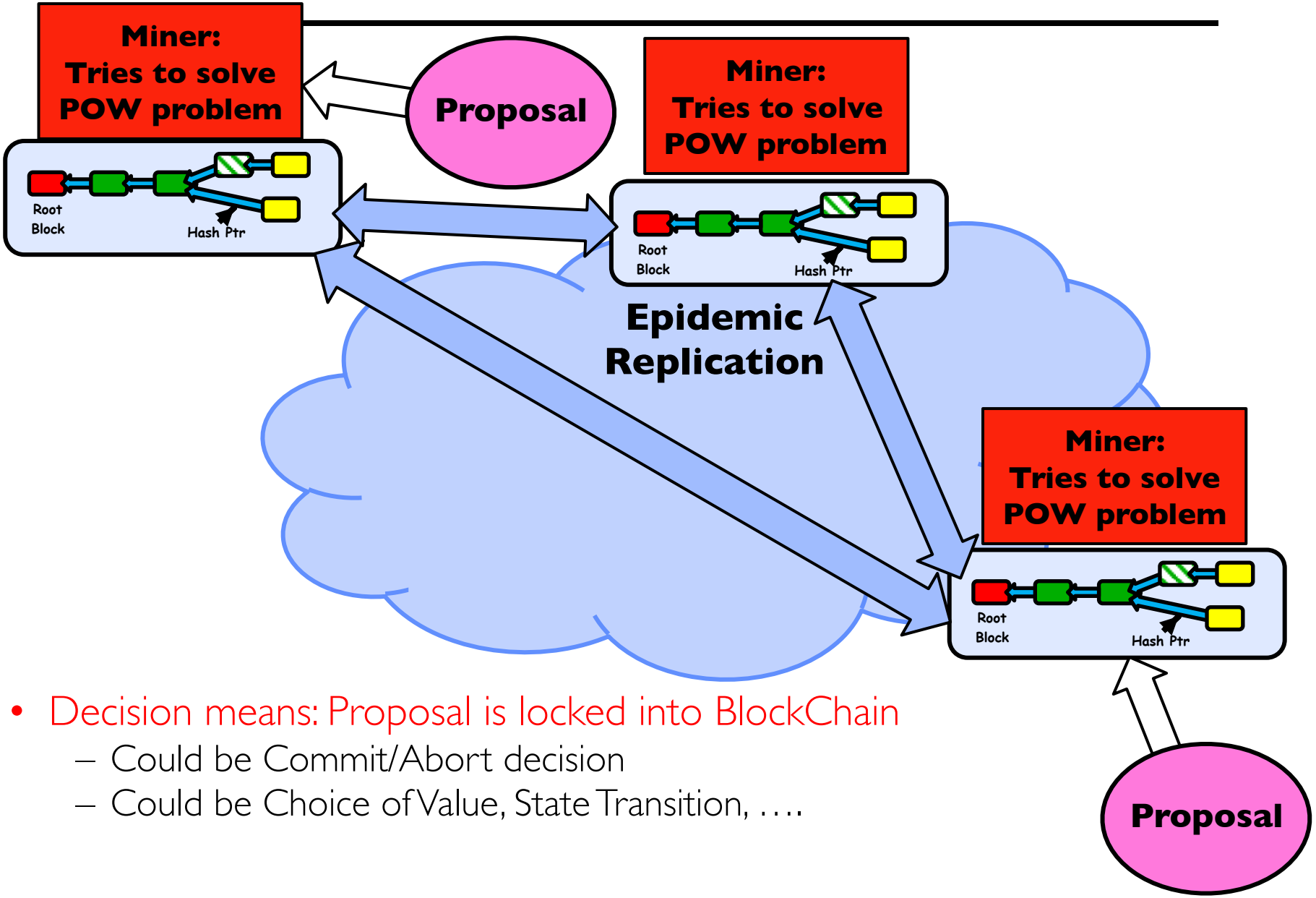
Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



Is a Blockchain a Distributed Decision Making Algorithm? (Con't)

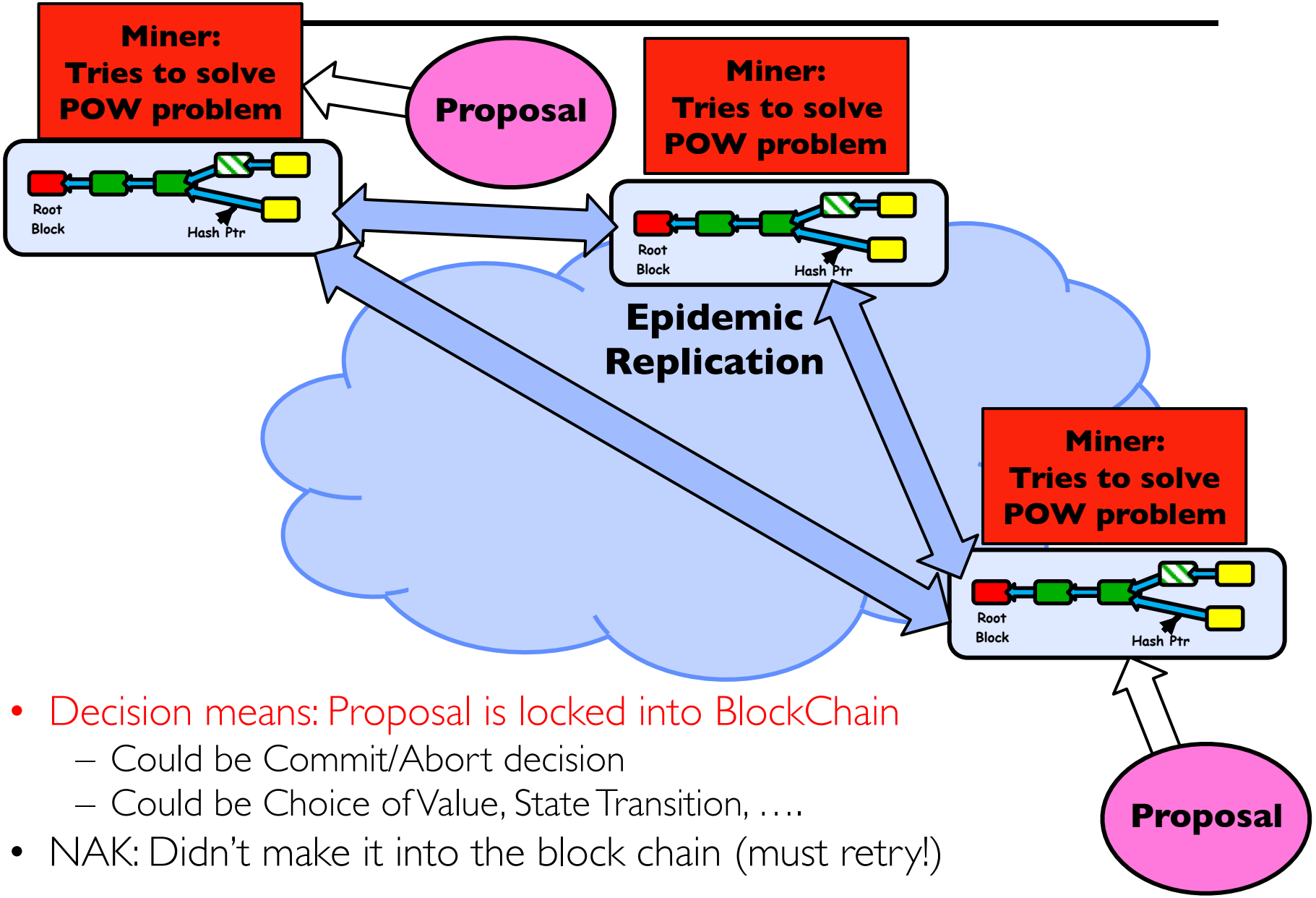


Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



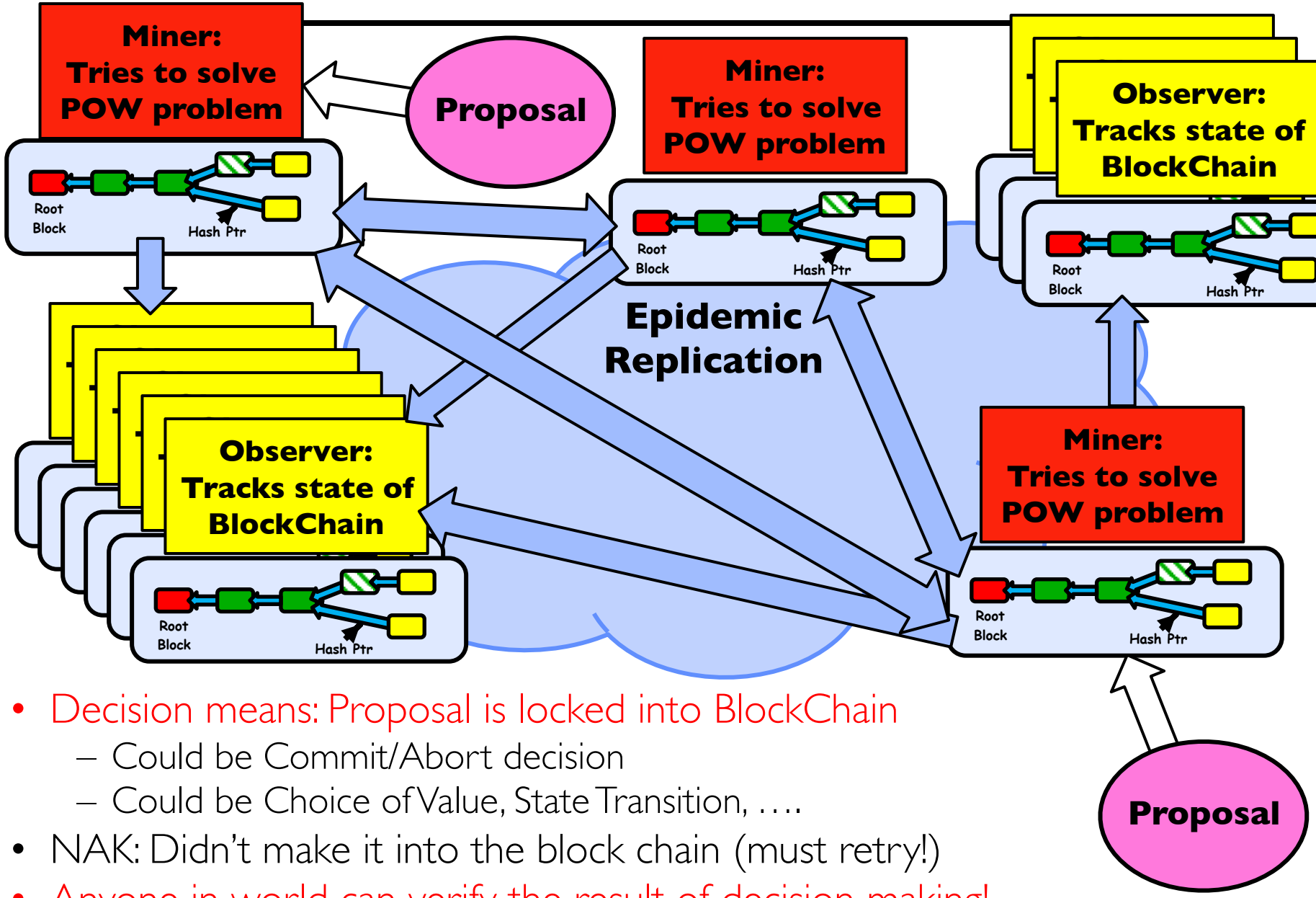
- Decision means: Proposal is locked into BlockChain
  - Could be Commit/Abort decision
  - Could be Choice of Value, State Transition, ….

Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



- Decision means: Proposal is locked into BlockChain
  - Could be Commit/Abort decision
  - Could be Choice of Value, State Transition, ….
- NAK: Didn't make it into the block chain (must retry!)

Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



- Decision means: Proposal is locked into BlockChain
  - Could be Commit/Abort decision
  - Could be Choice of Value, State Transition, ….
- NAK: Didn't make it into the block chain (must retry!)
- Anyone in world can verify the result of decision making!

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 – what about machines with different byte order ("BigEndian" vs "LittleEndian")

- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    ```
    remoteFileSystem→Read("rutabaga");
    ```
  - Translated automatically into call on server:
    ```
    fileSys→Read("rutabaga");
    ```

RPC Concept

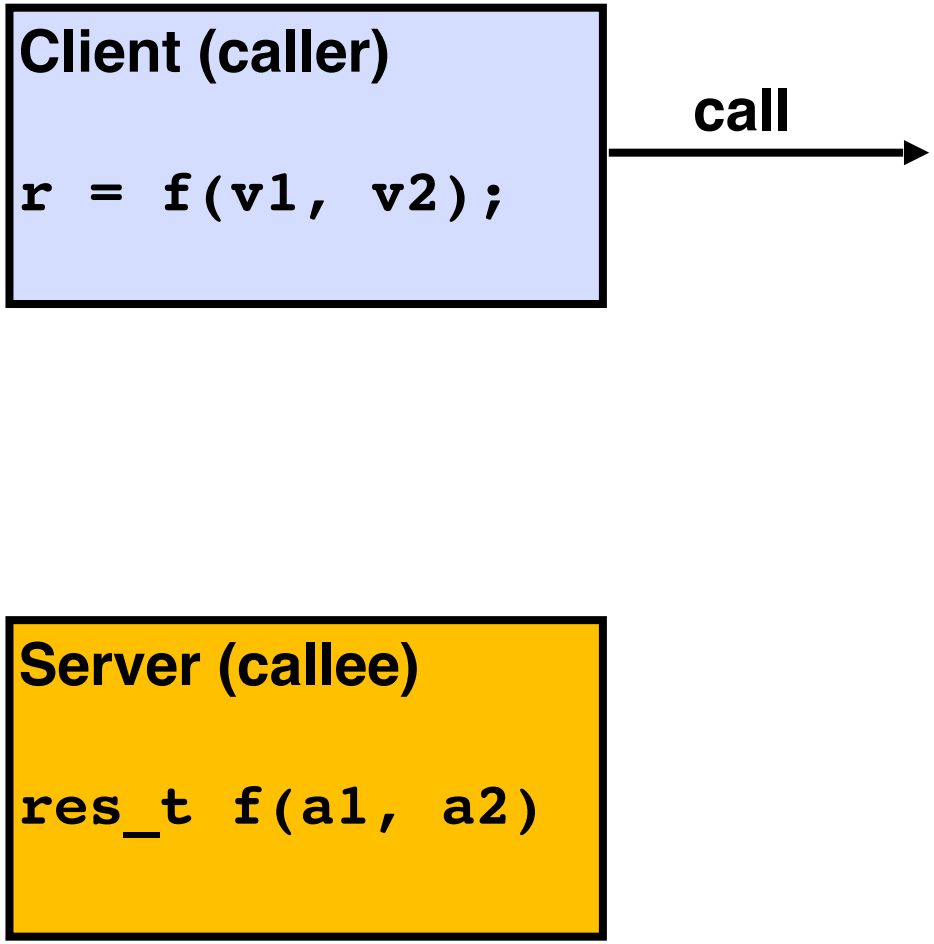
Client (caller)

```
r = f(v1, v2);
```

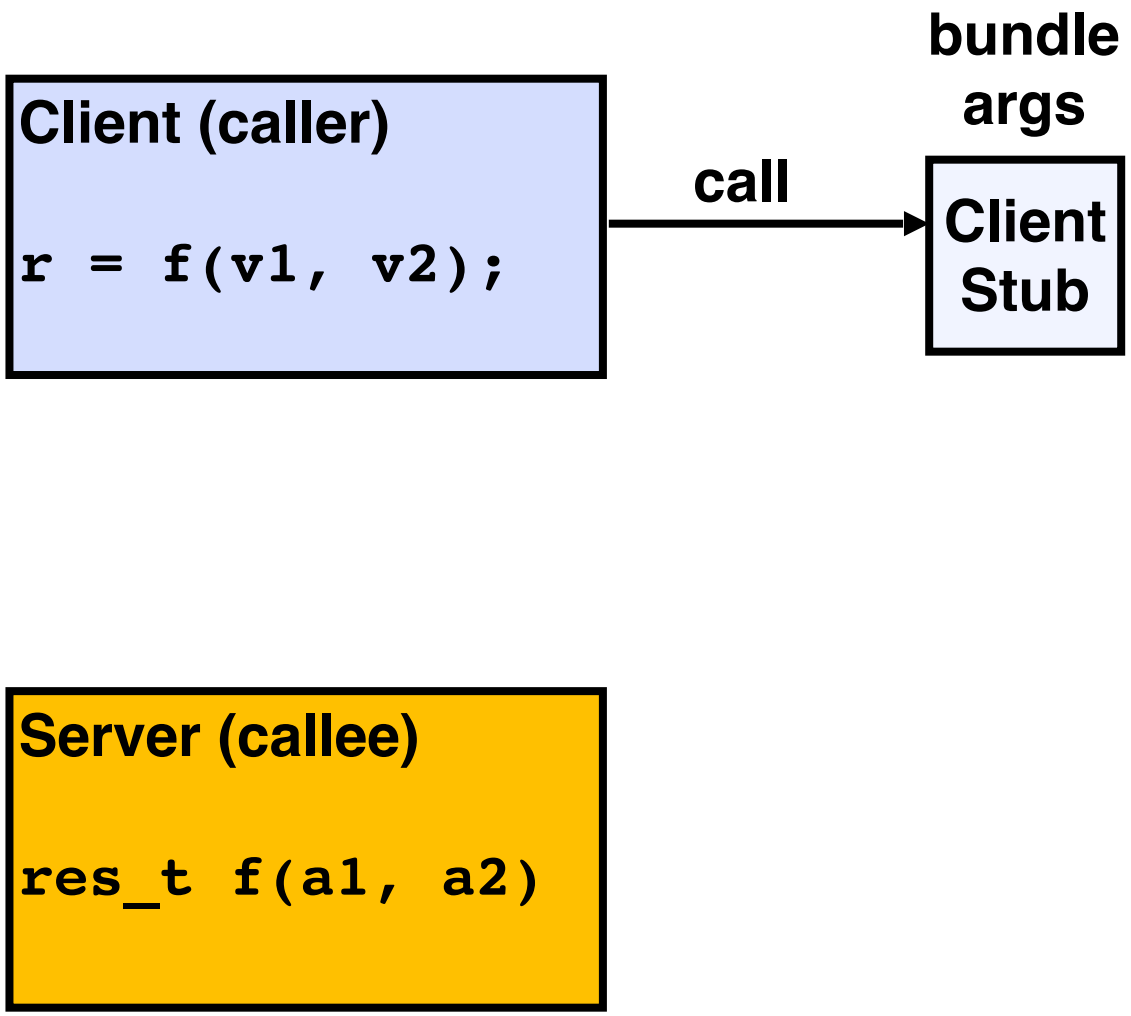
Server (callee)

```
res_t f(a1, a2)
```

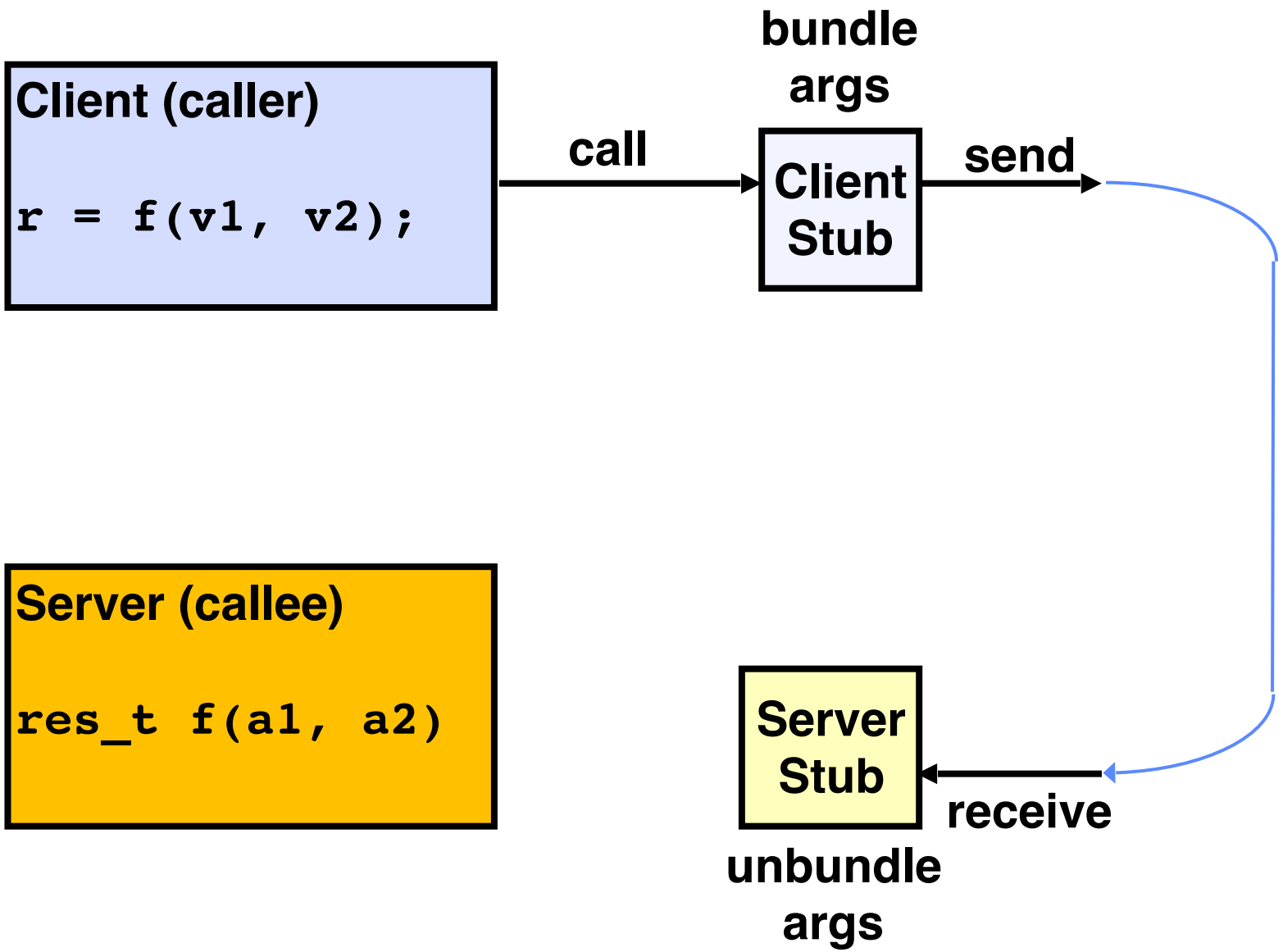
RPC Concept



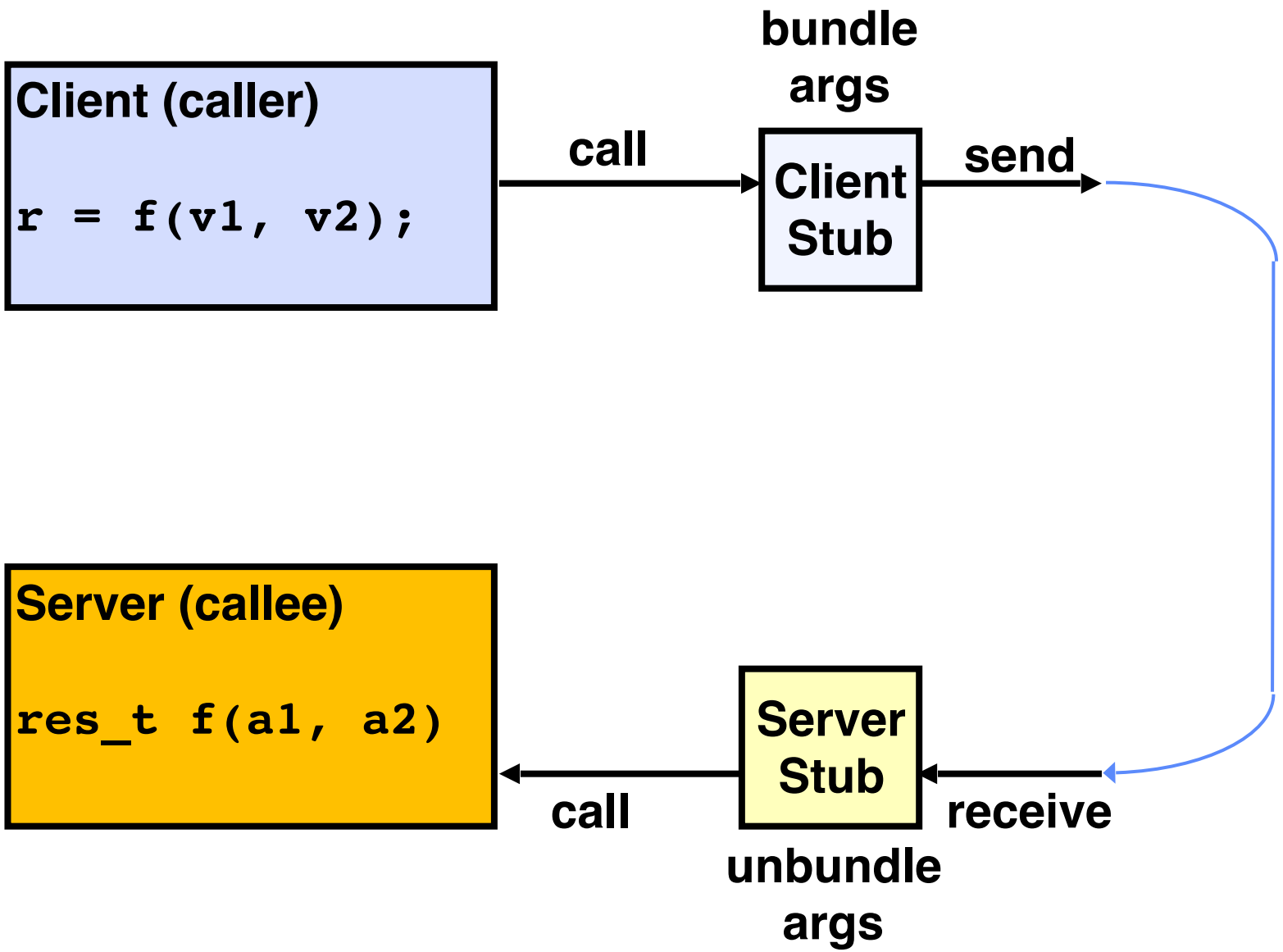
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
Server (callee)
res_t f(a1, a2)

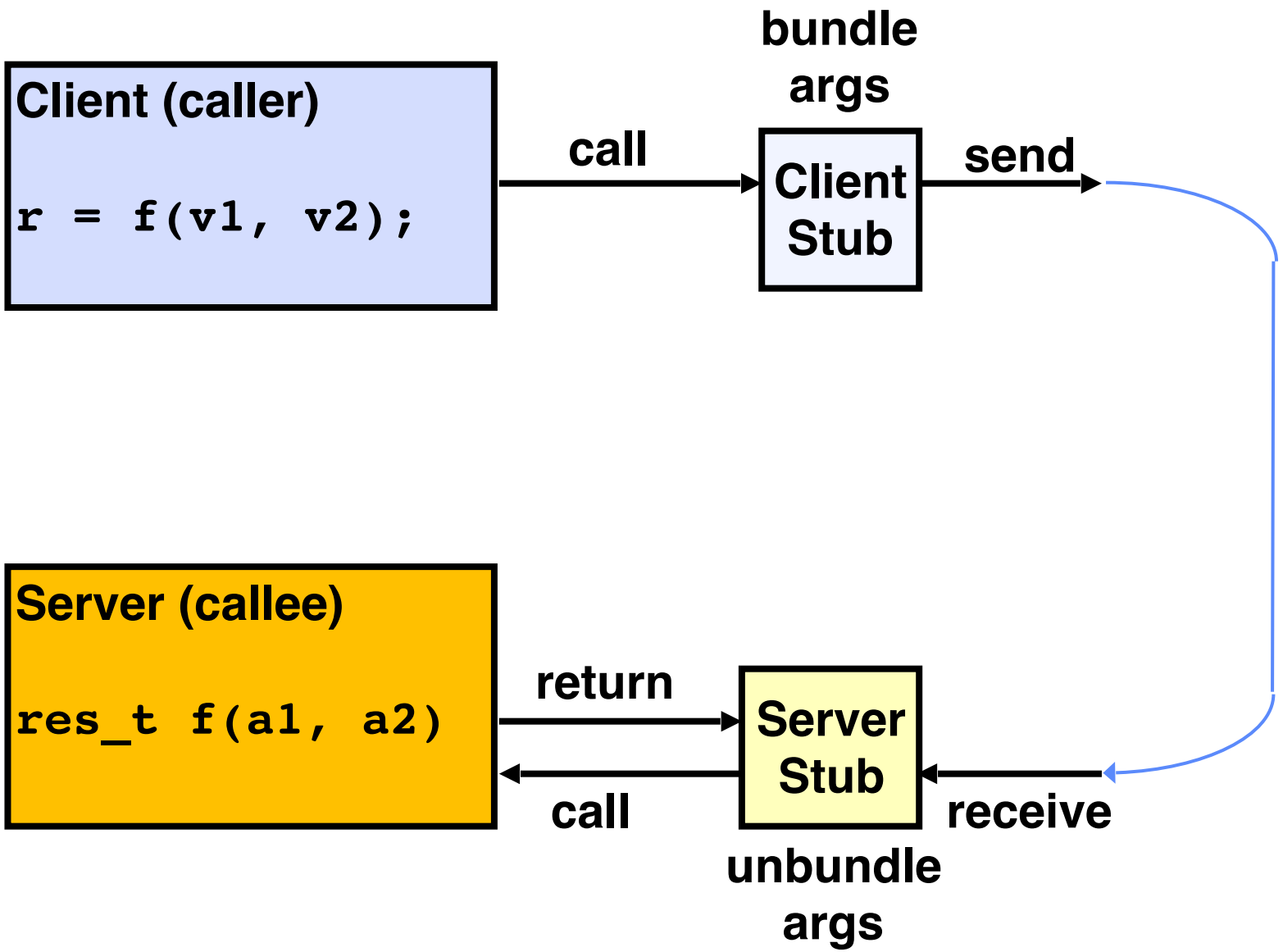
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
send
Server (callee)
res_t f(a1, a2)
Server
Stub
receive
unbundle
args

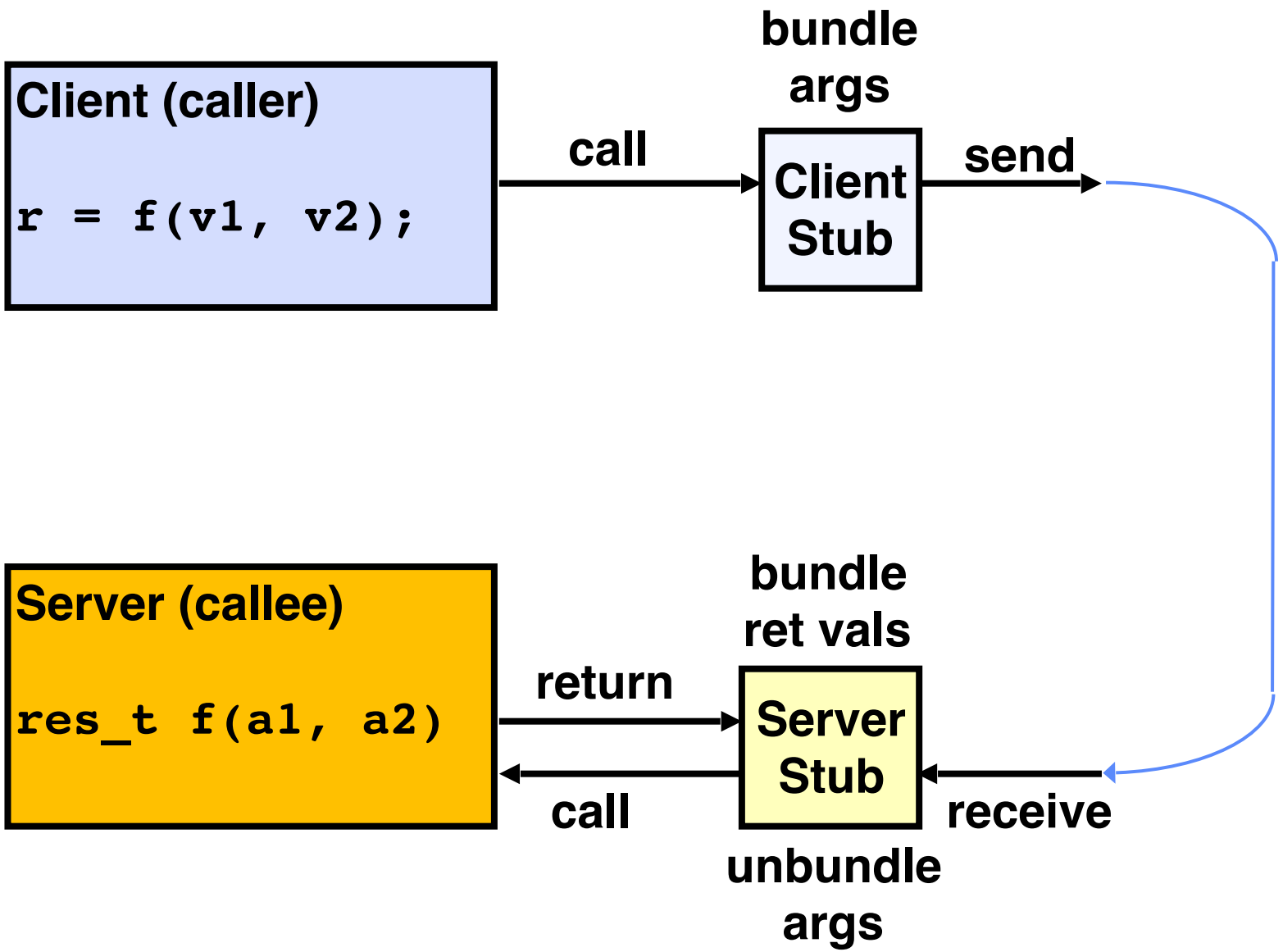
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
send
Server (callee)
res_t f(a1, a2)
call
Server
Stub
receive
unbundle
args

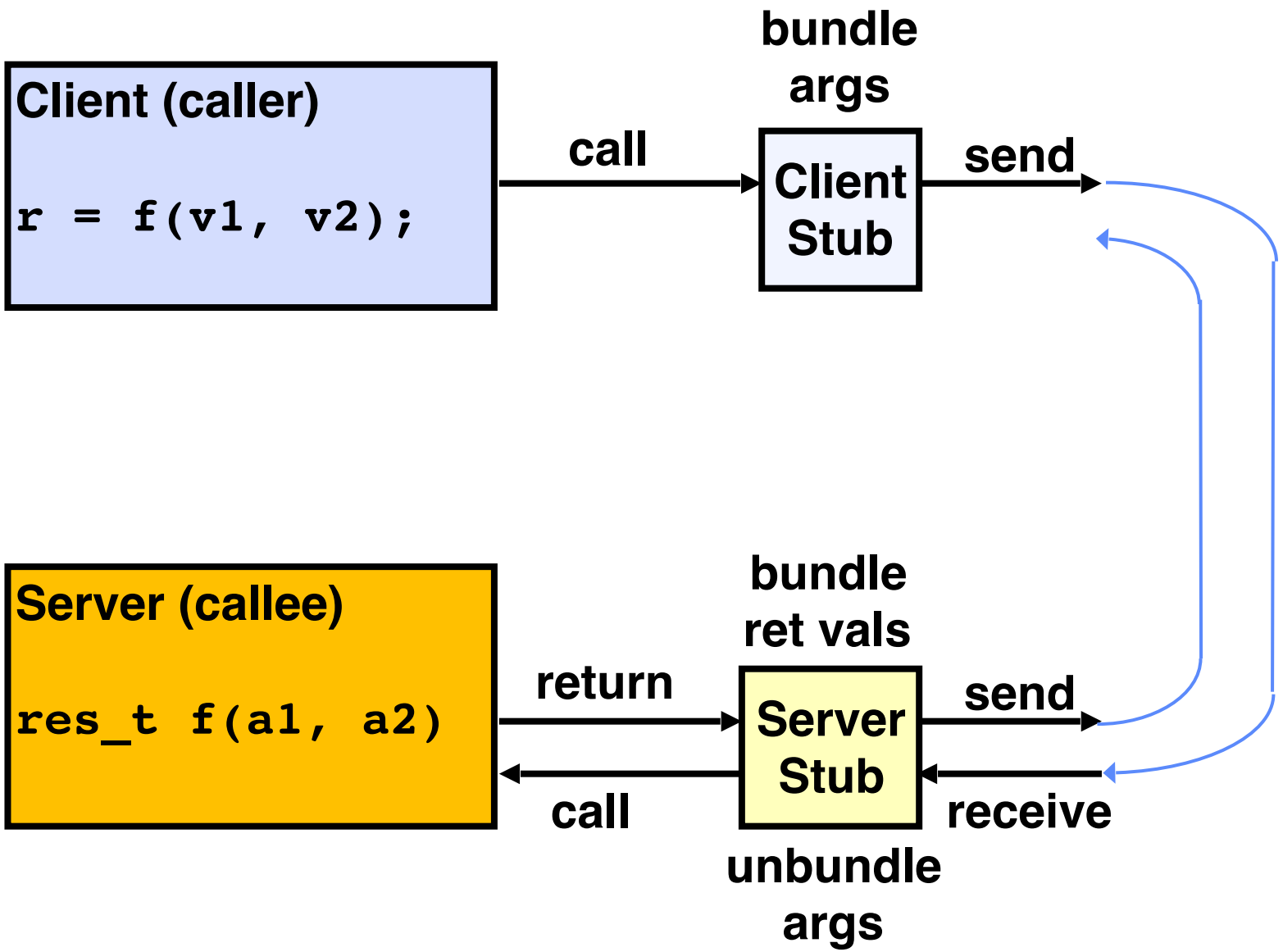
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle args
Client Stub
send
Server (callee)
res_t f(a1, a2)
return
call
Server Stub
receive
unbundle args

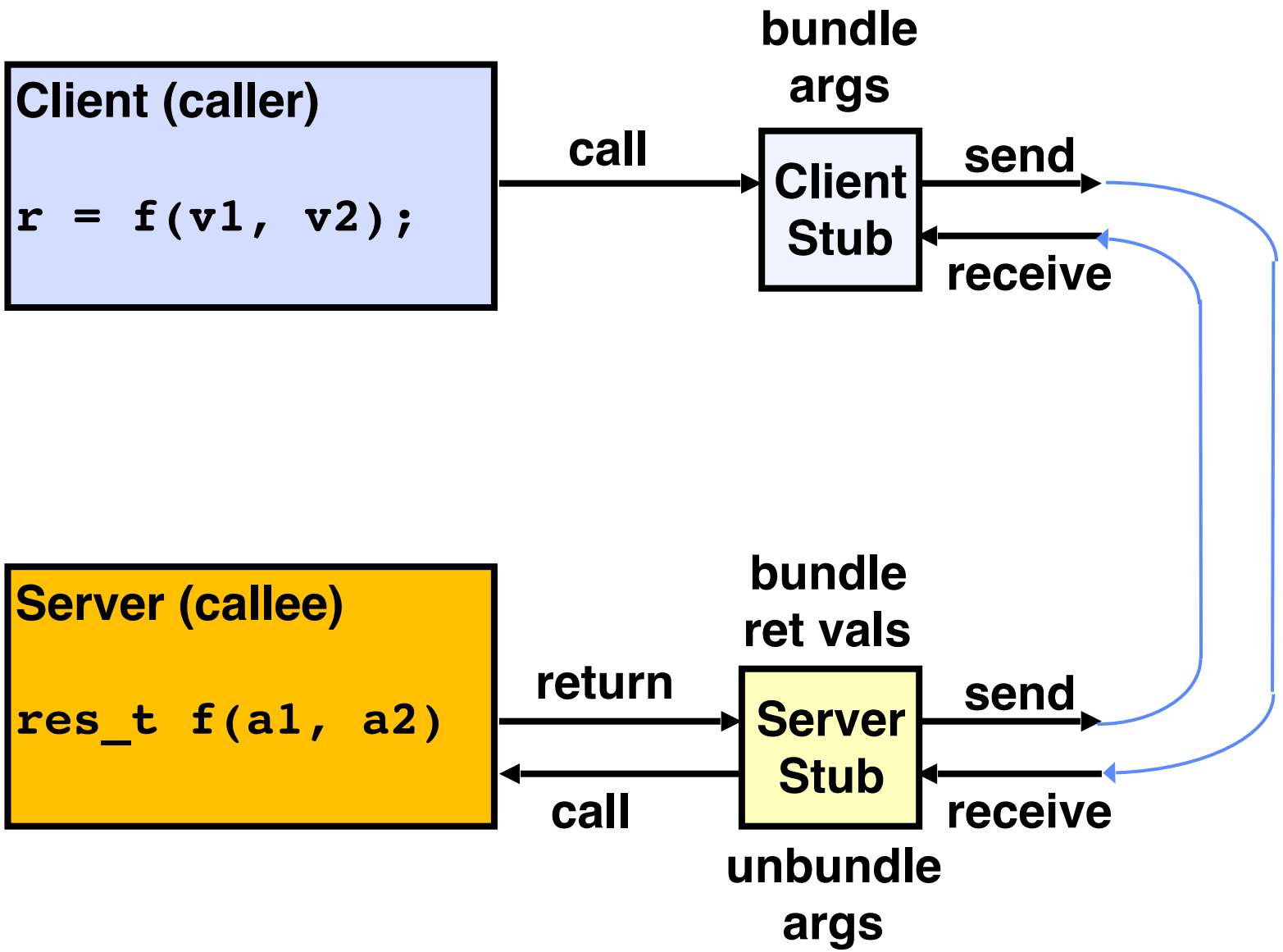
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
send
Server (callee)
res_t f(a1, a2)
return
bundle
ret vals
Server
Stub
call
receive
unbundle
args

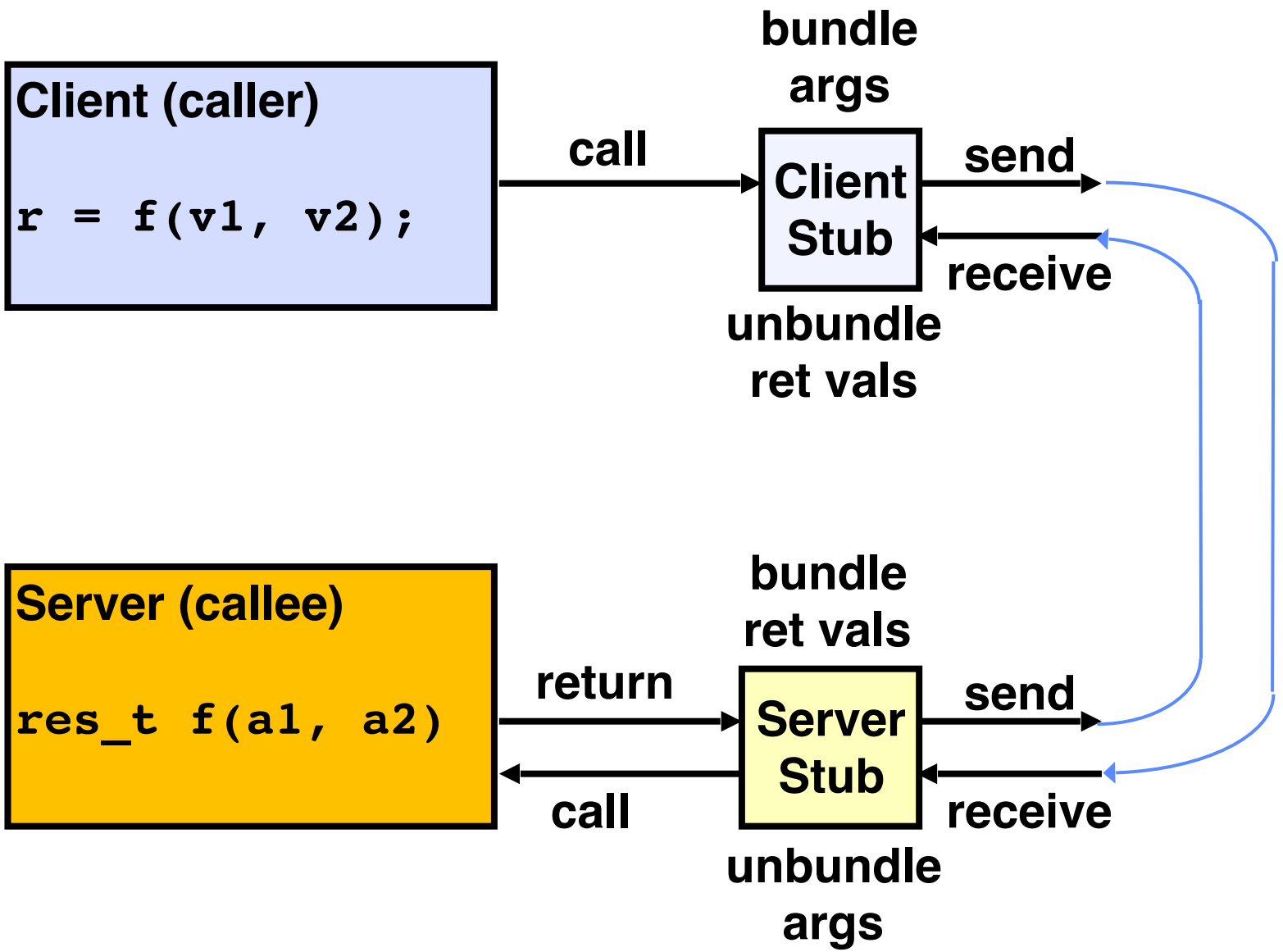
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
send
Server (callee)
res_t f(a1, a2)
return
bundle
ret vals
Server
Stub
send
call
receive
unbundle
args

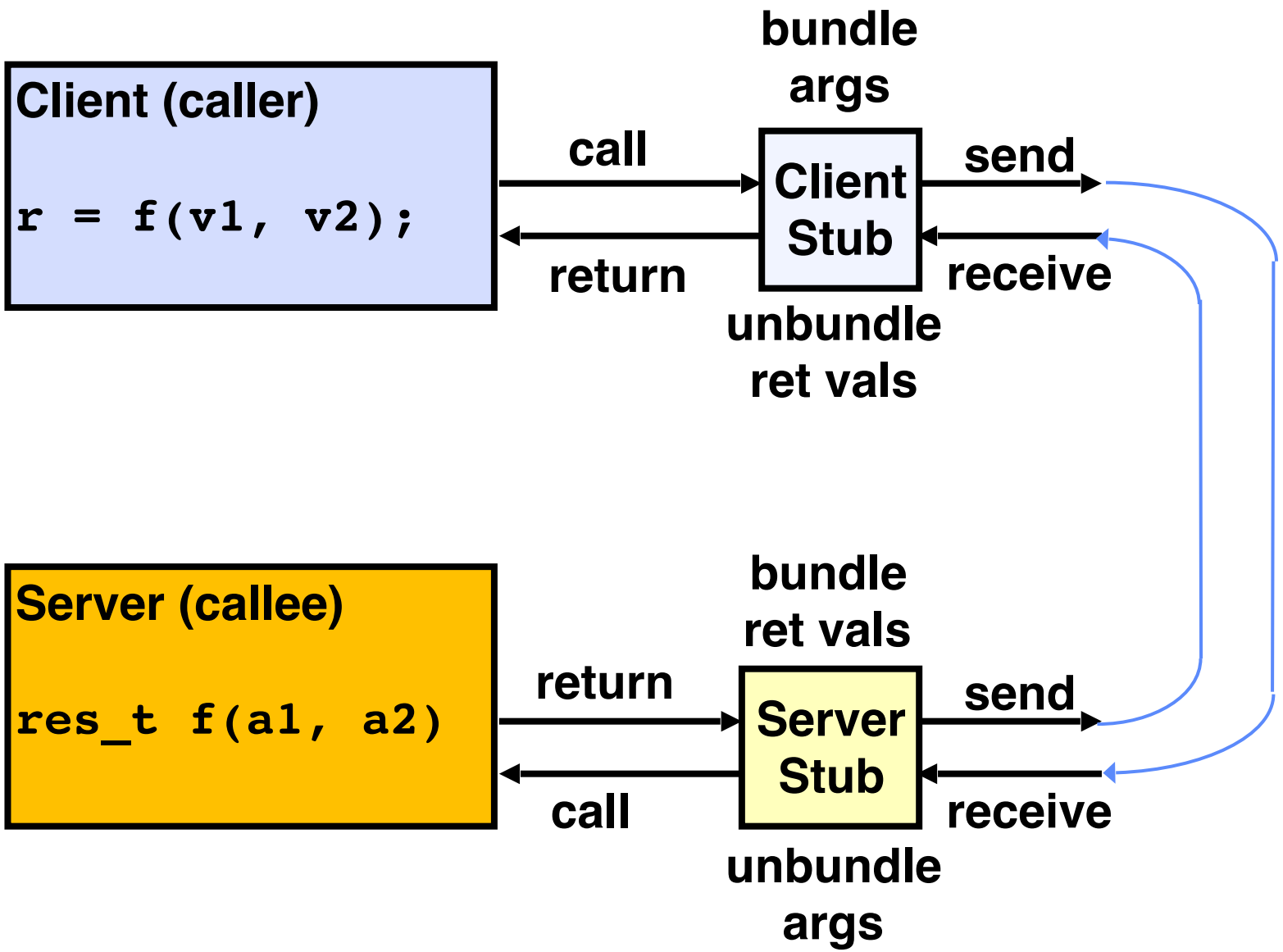
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
send
receive
Server (callee)
res_t f(a1, a2)
return
call
bundle
ret vals
Server
Stub
send
receive
unbundle
args

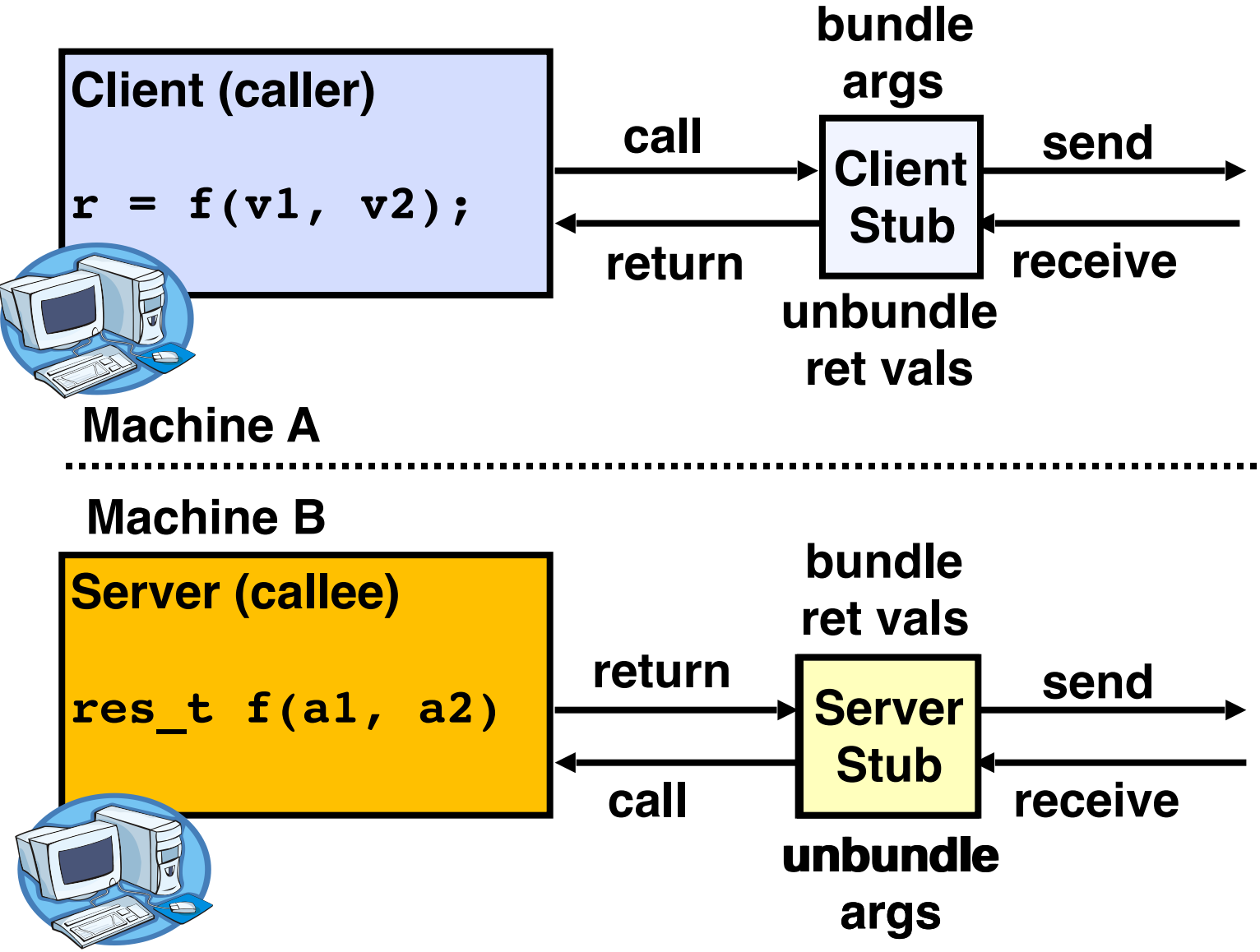
RPC Concept


Client (caller)
r = f(v1, v2);
call
bundle
args
Client
Stub
send
receive
unbundle
ret vals
Server (callee)
res_t f(a1, a2)
return
bundle
ret vals
Server
Stub
send
call
receive
unbundle
args

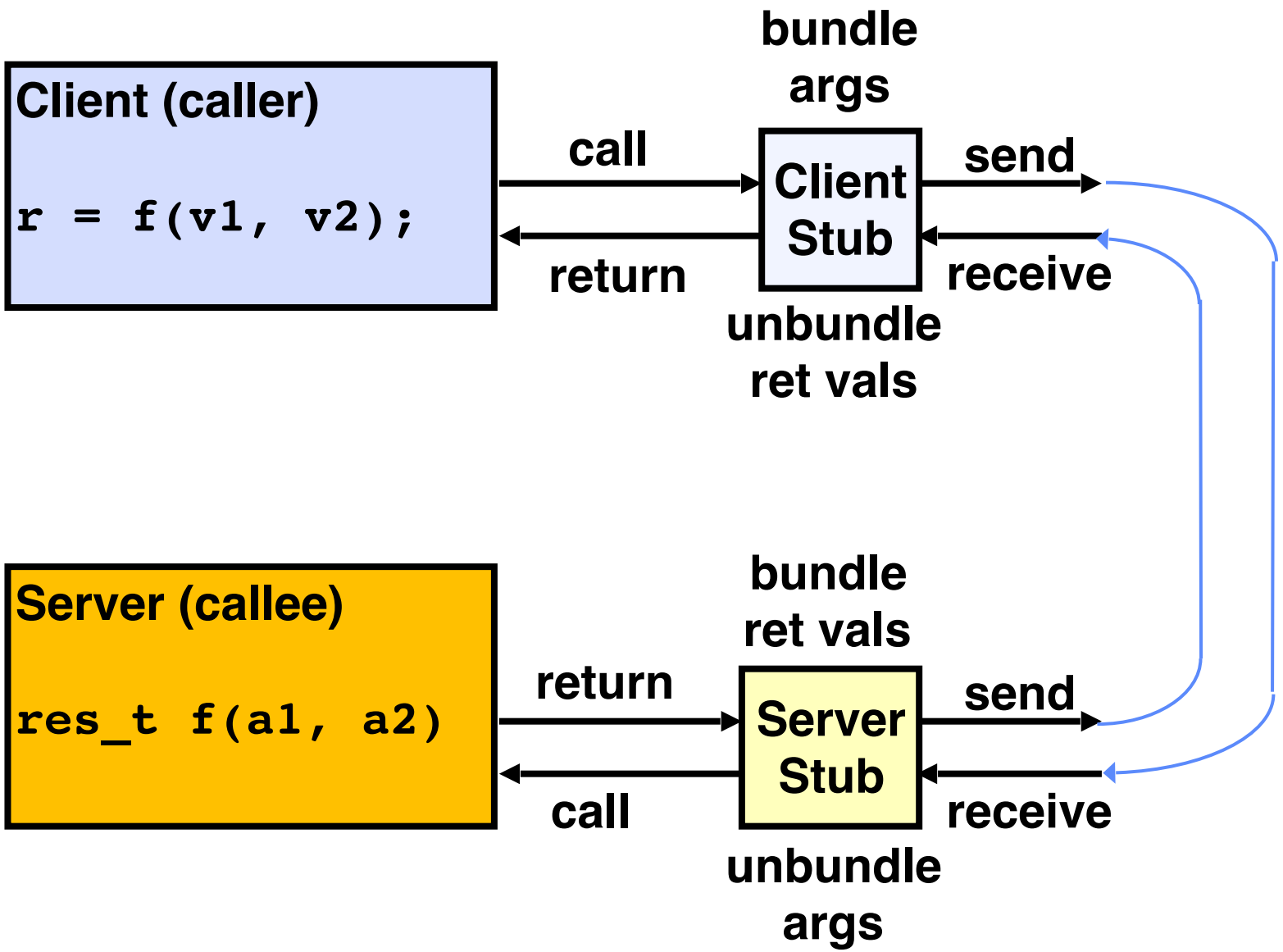
RPC Concept


Client (caller)
r = f(v1, v2);
bundle
args
call
Client
Stub
send
return
receive
unbundle
ret vals
Server (callee)
res_t f(a1, a2)
bundle
ret vals
return
Server
Stub
send
call
receive
unbundle
args

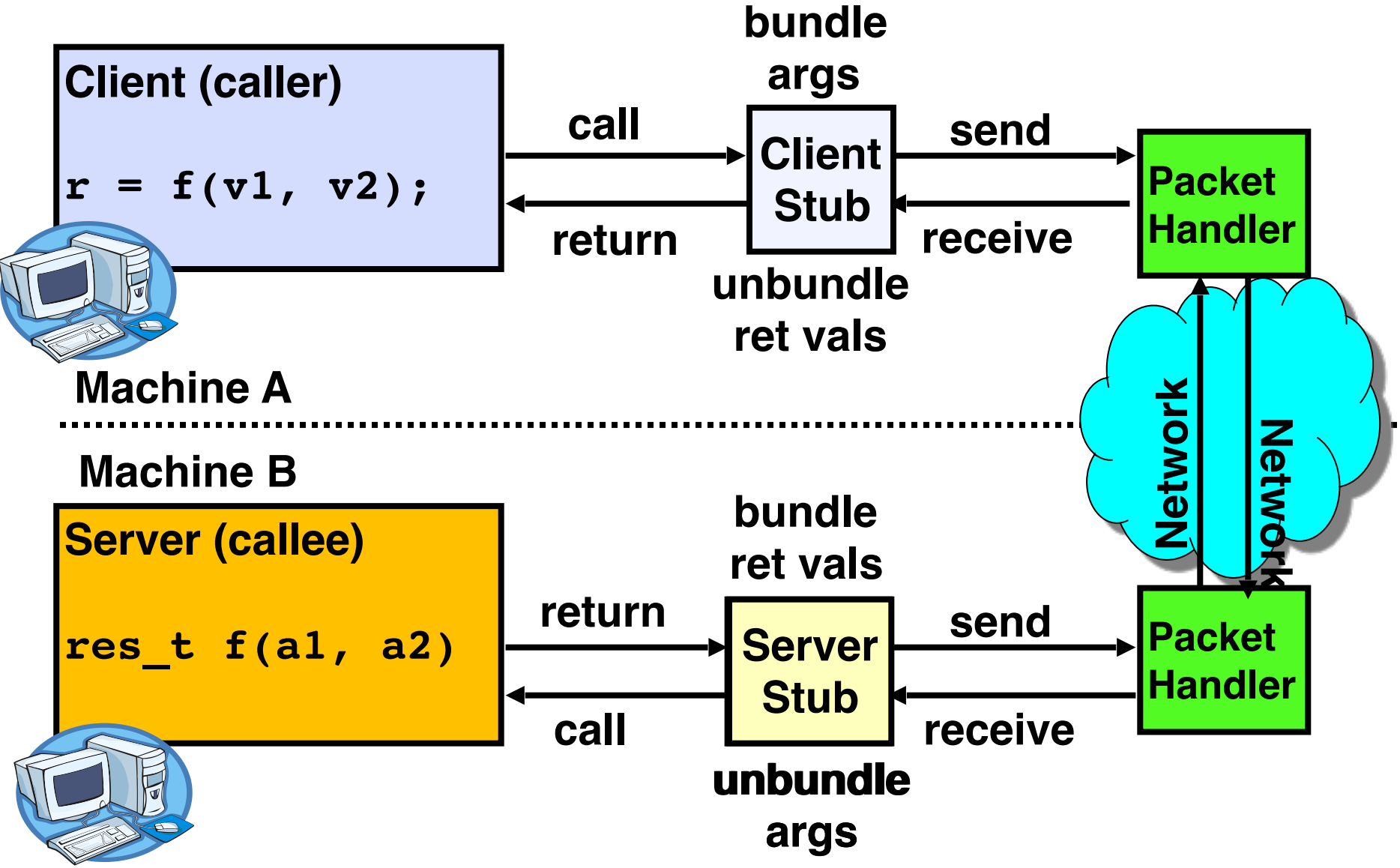
RPC Information Flow



RPC Concept


Client (caller)
r = f(v1, v2);
bundle
args
call
Client
Stub
send
return
receive
unbundle
ret vals
Server (callee)
res_t f(a1, a2)
bundle
ret vals
return
Server
Stub
send
call
receive
unbundle
args

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 ⇔ 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 (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 → 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 « same-machine RPC « network RPC
  - Overheads: Marshalling, Stubs, Kernel-Crossing, Communication

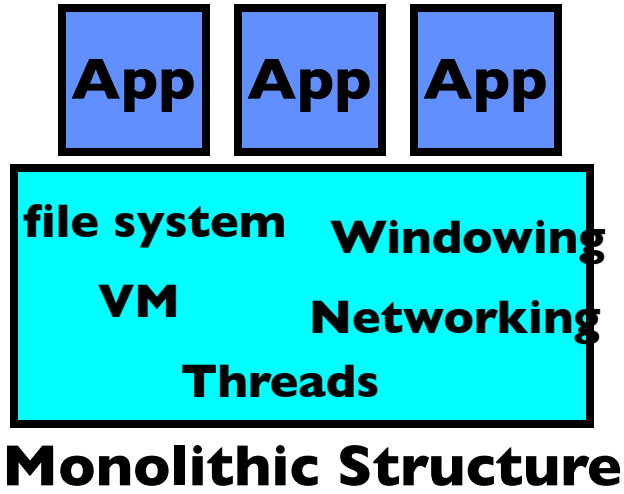
- Programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex

Cross-Domain Communication / Location Transparency

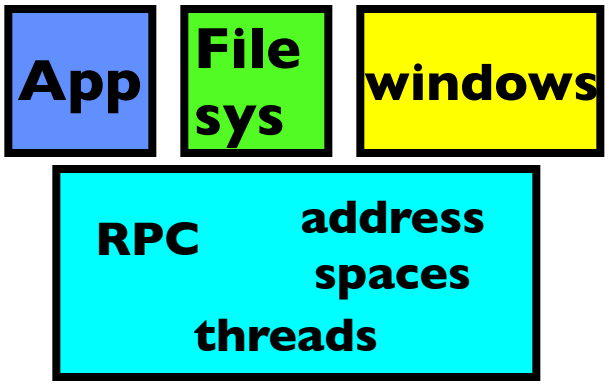
- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - "Remote" procedure call (2-way communication)
- RPC's can be used to communicate between address spaces on different machines or the same machine
  - 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



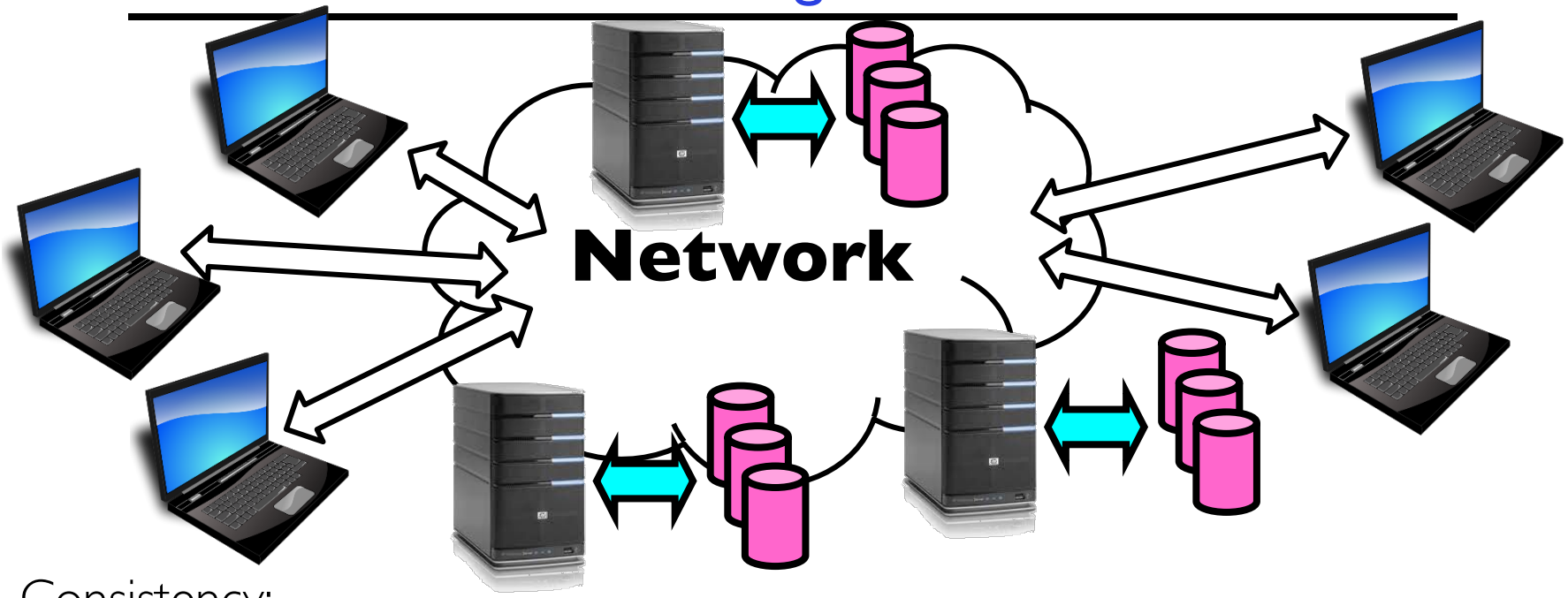
Monolithic Structure



Microkernel Structure

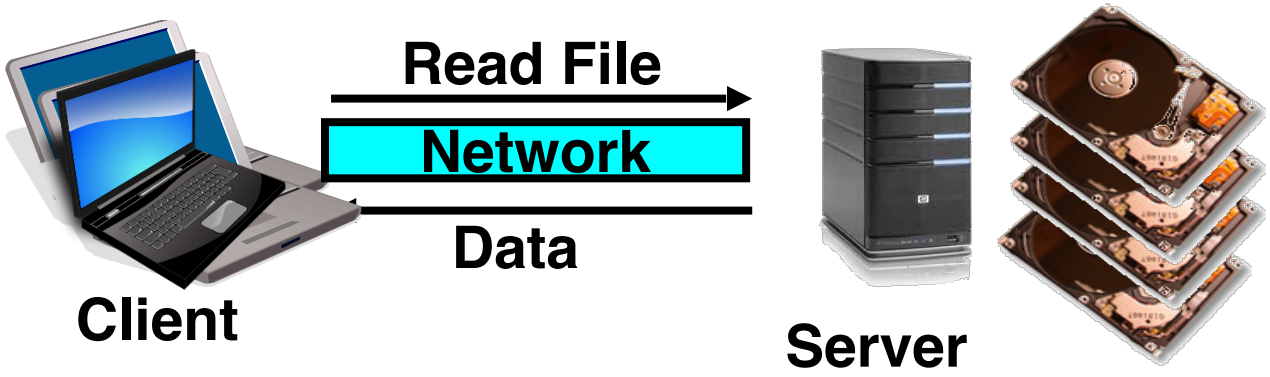
- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: Each X client can be on a separate machine from X server;

Network-Attached Storage and the CAP Theorem



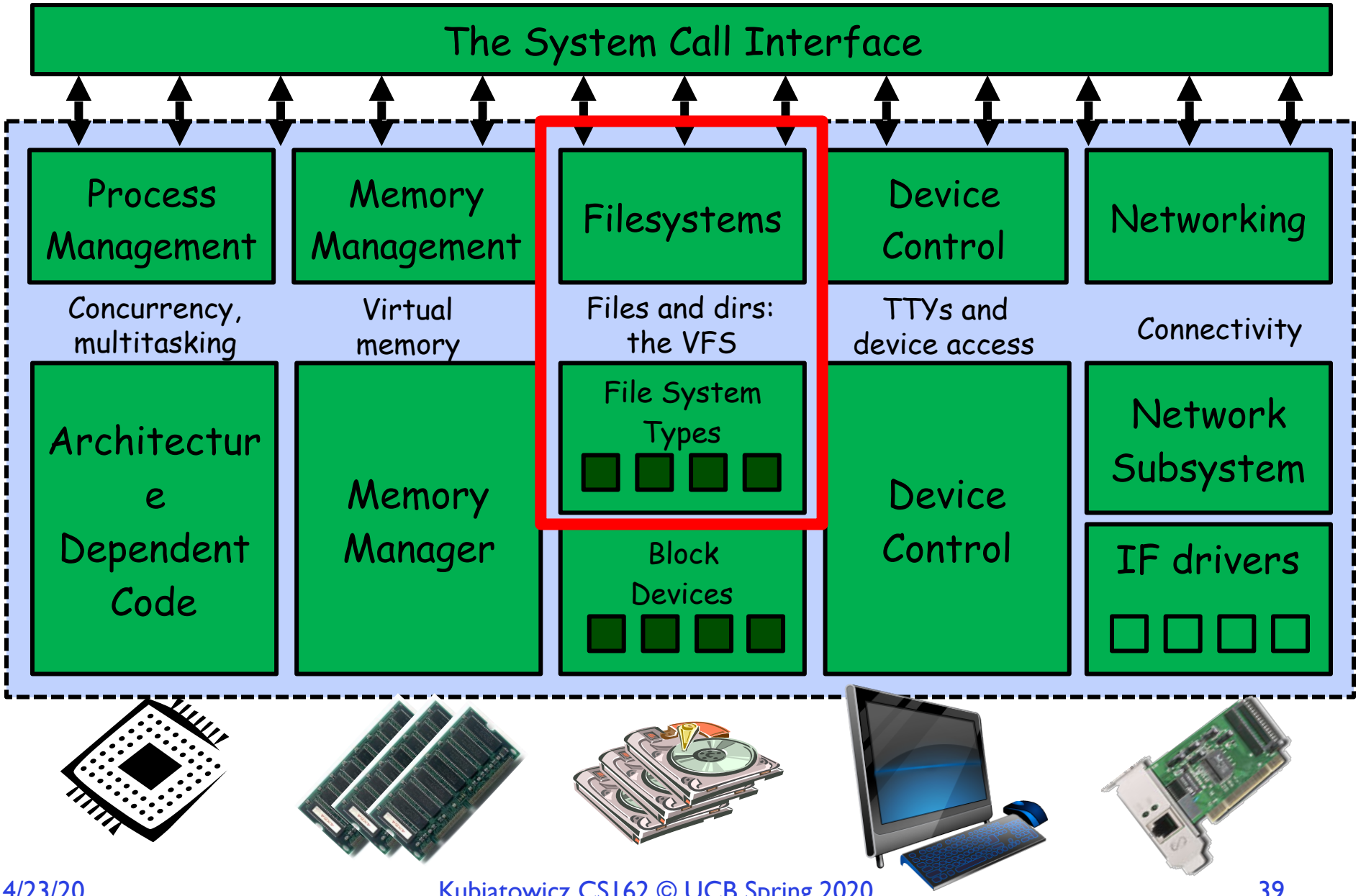
- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same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

Distributed File Systems

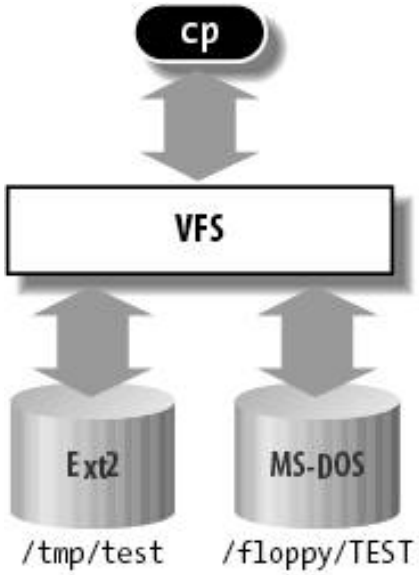


- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/home/oksi/162/` on laptop actually refers to `/users/oski` on campus file server

Enabling Design:VFS



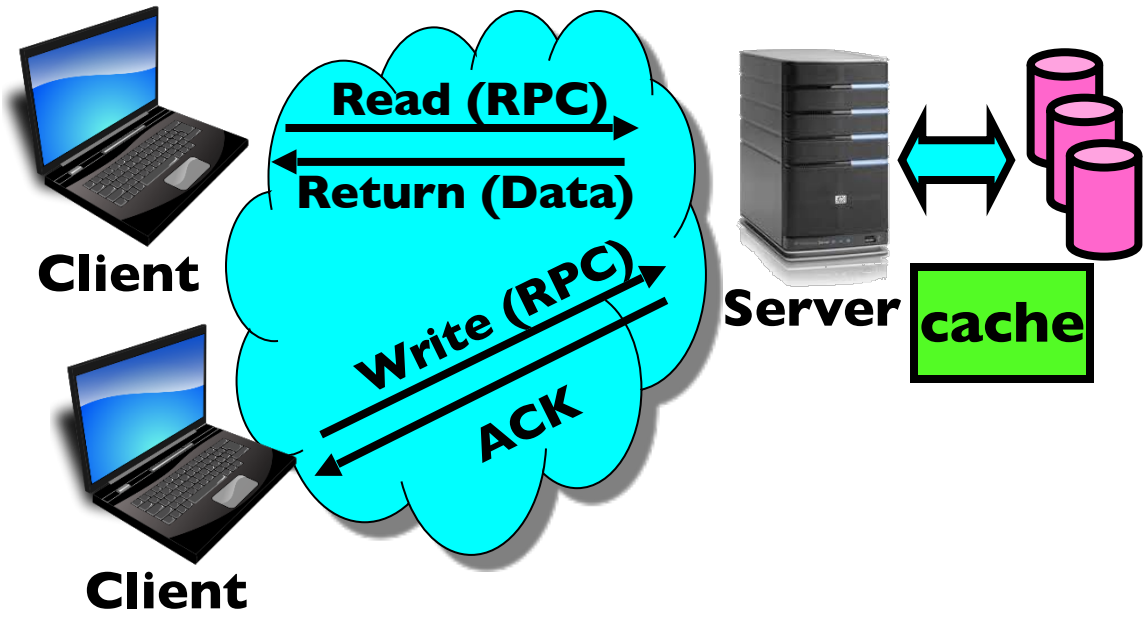
Virtual Filesystem Switch (Con't)




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

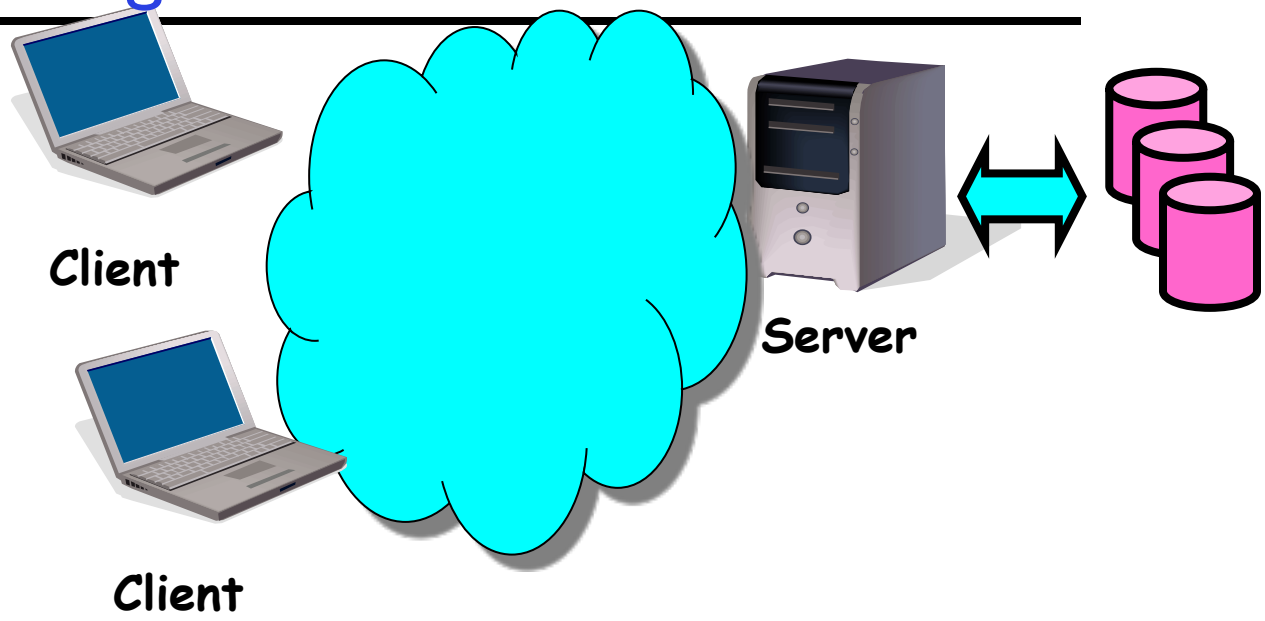
- VFS: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Simple Distributed File System

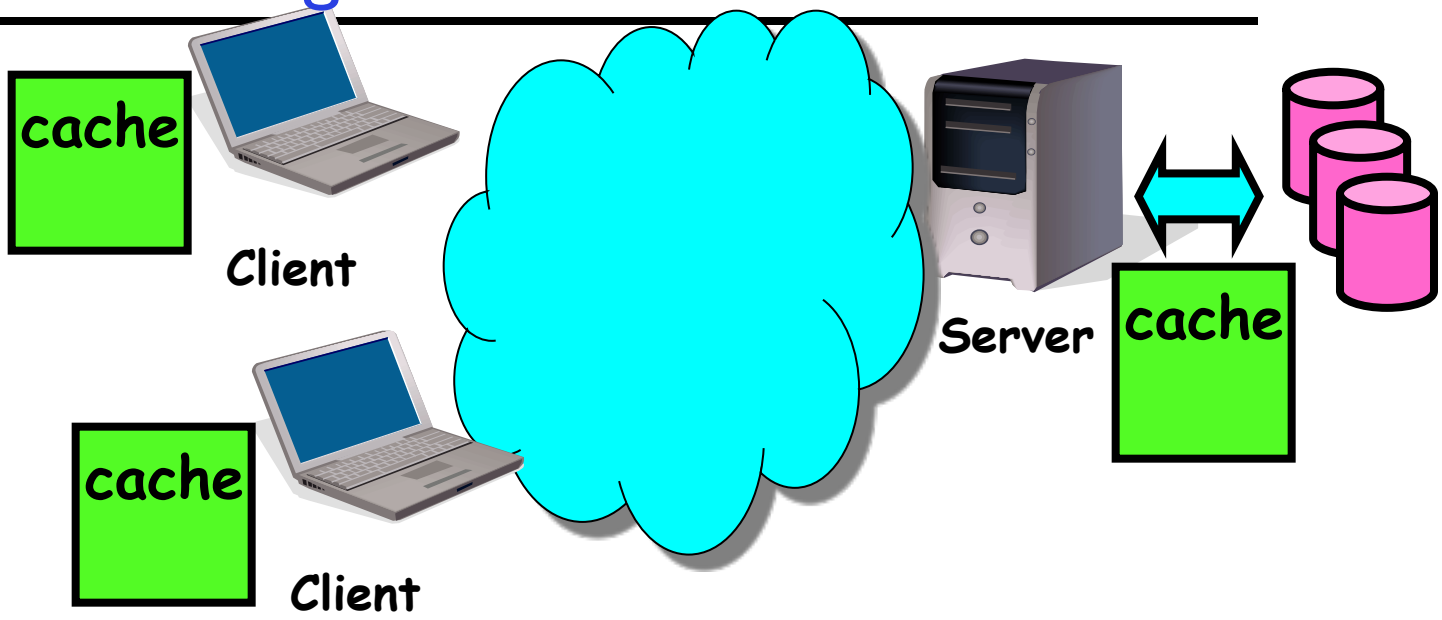


- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic
  - Server can be a bottleneck

Use of caching to reduce network load

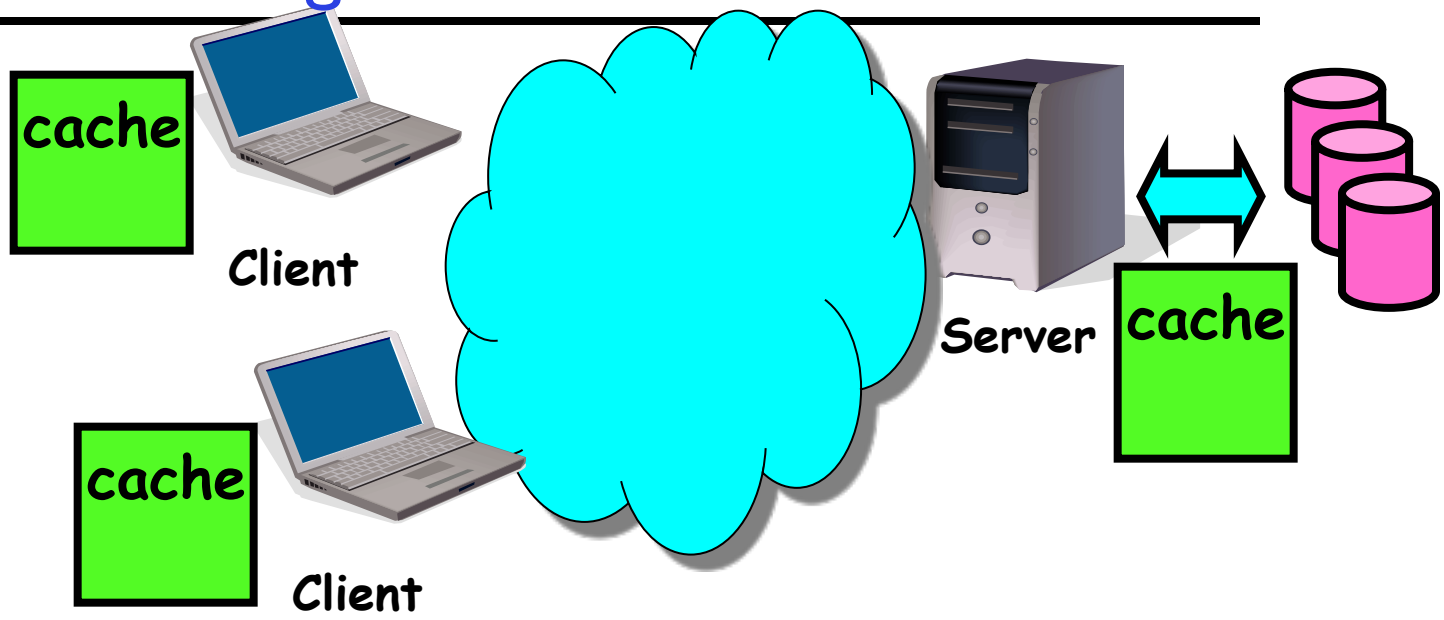


Use of caching to reduce network load

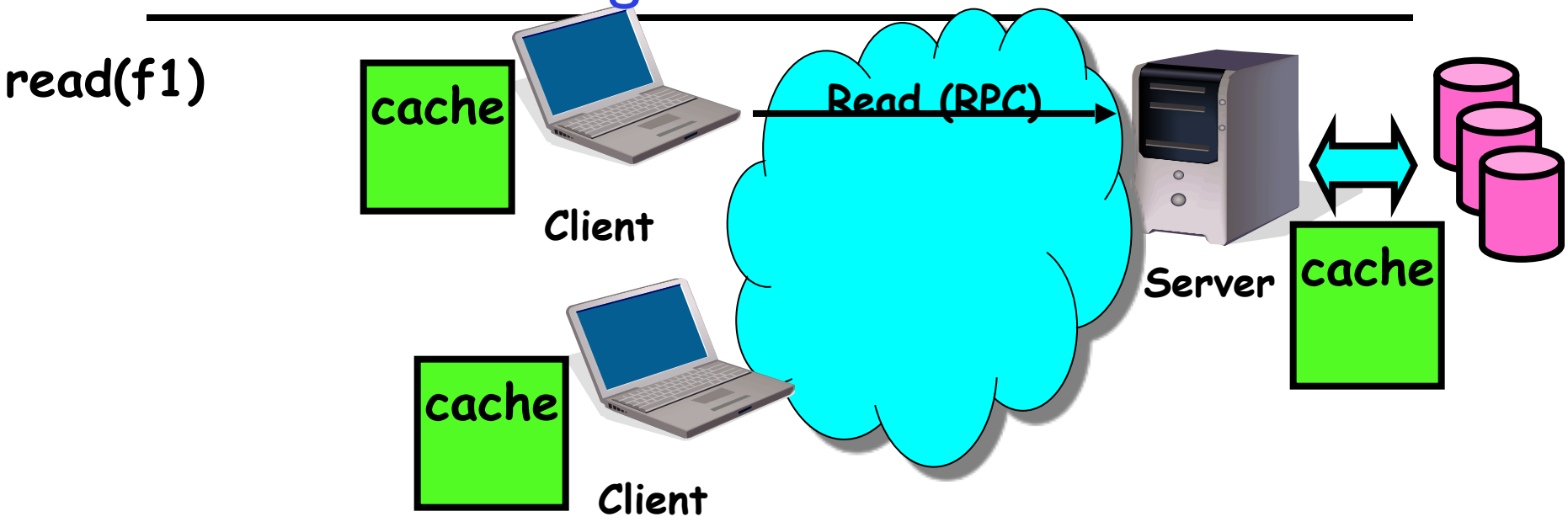


Use of caching to reduce network load

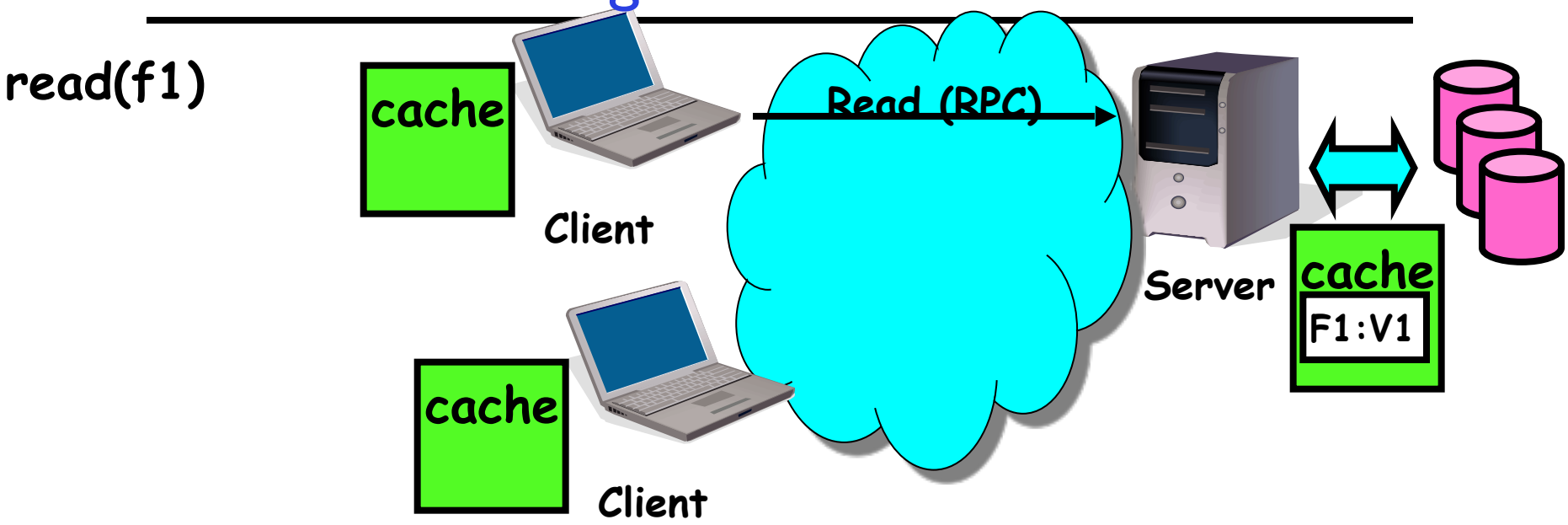
read(f1)



Use of caching to reduce network load

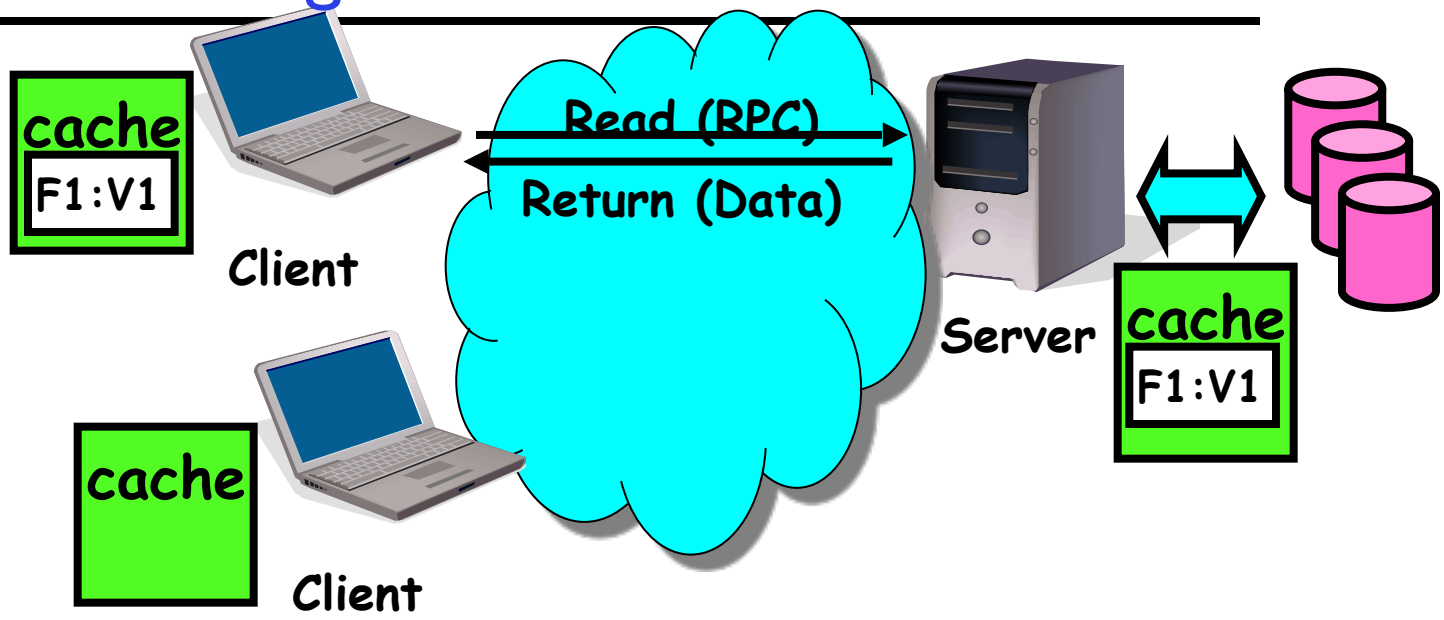


Use of caching to reduce network load

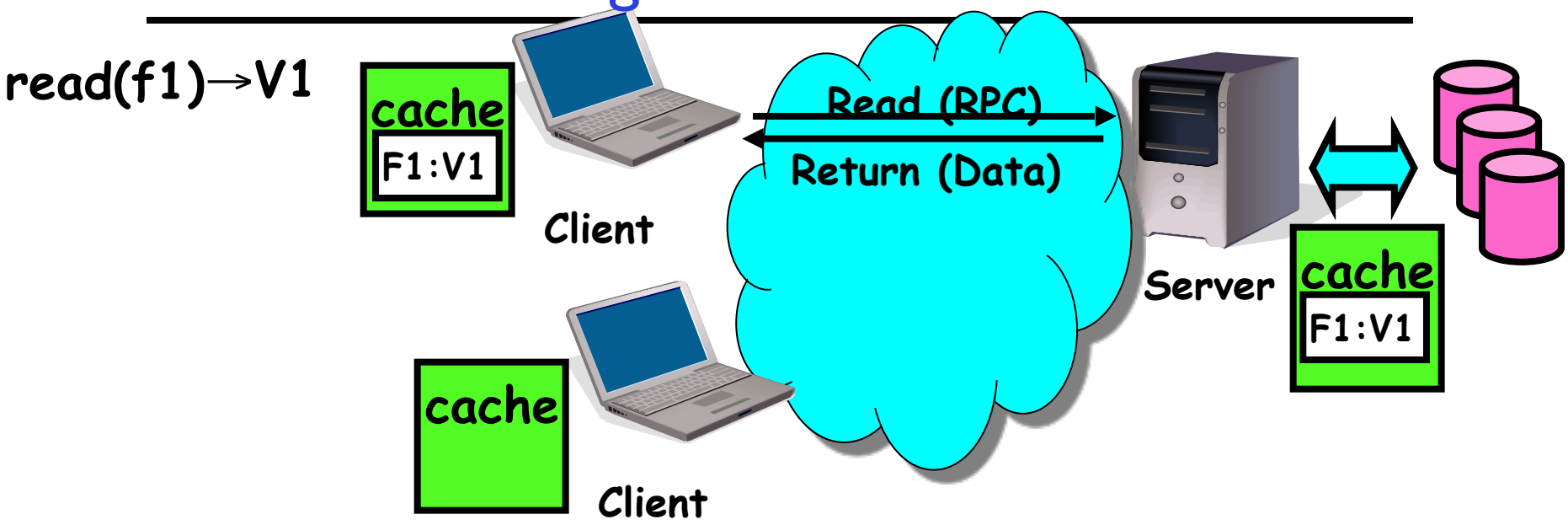


Use of caching to reduce network load

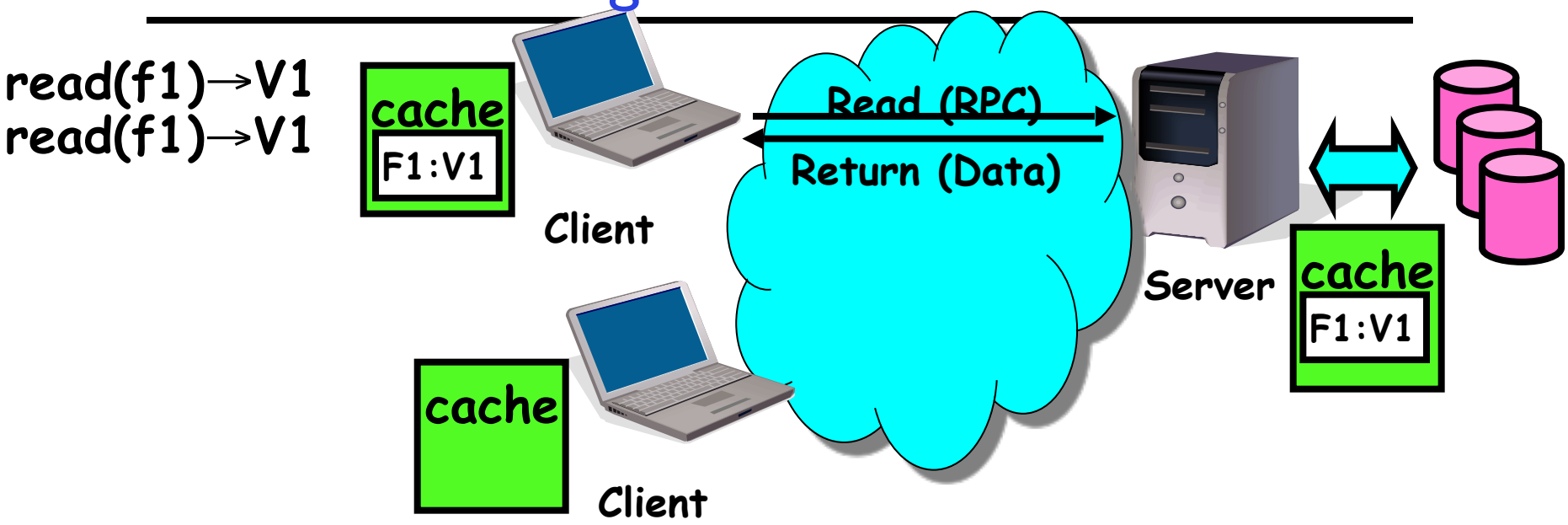
read(f1)



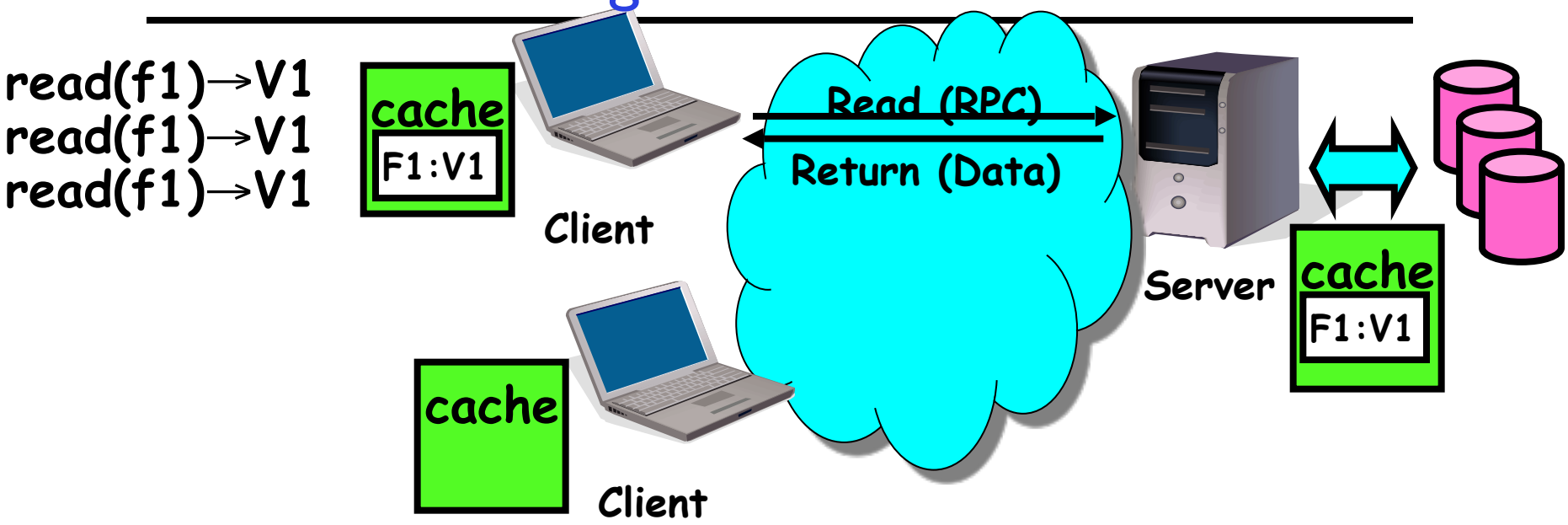
Use of caching to reduce network load



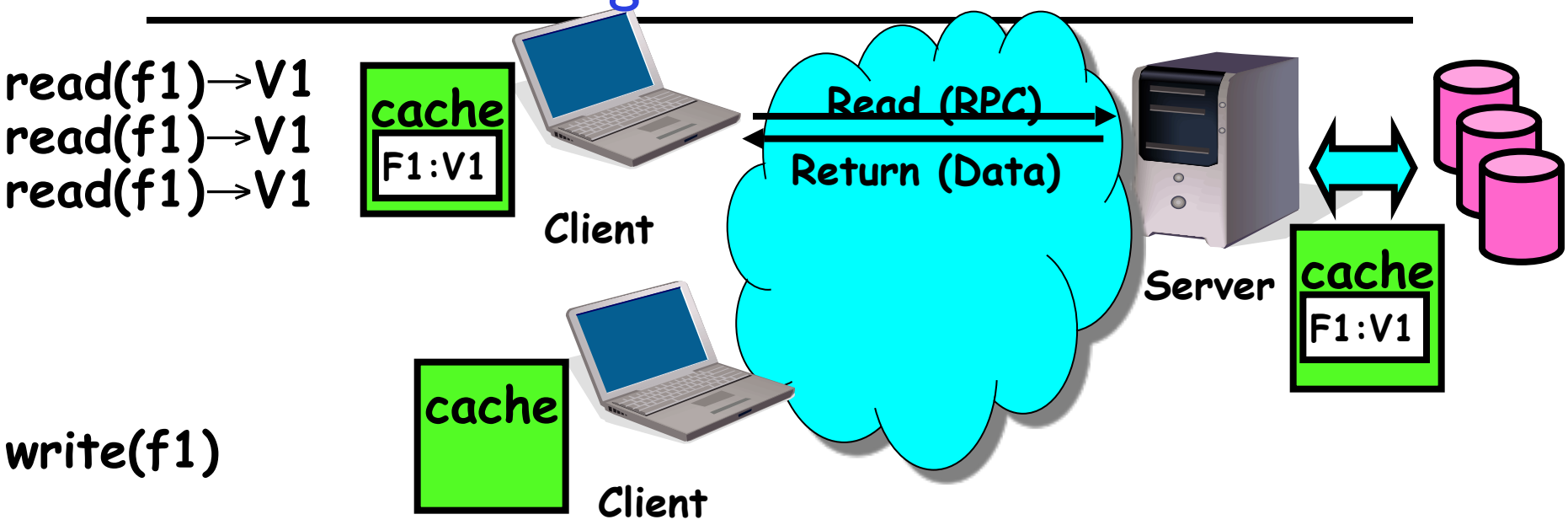
Use of caching to reduce network load



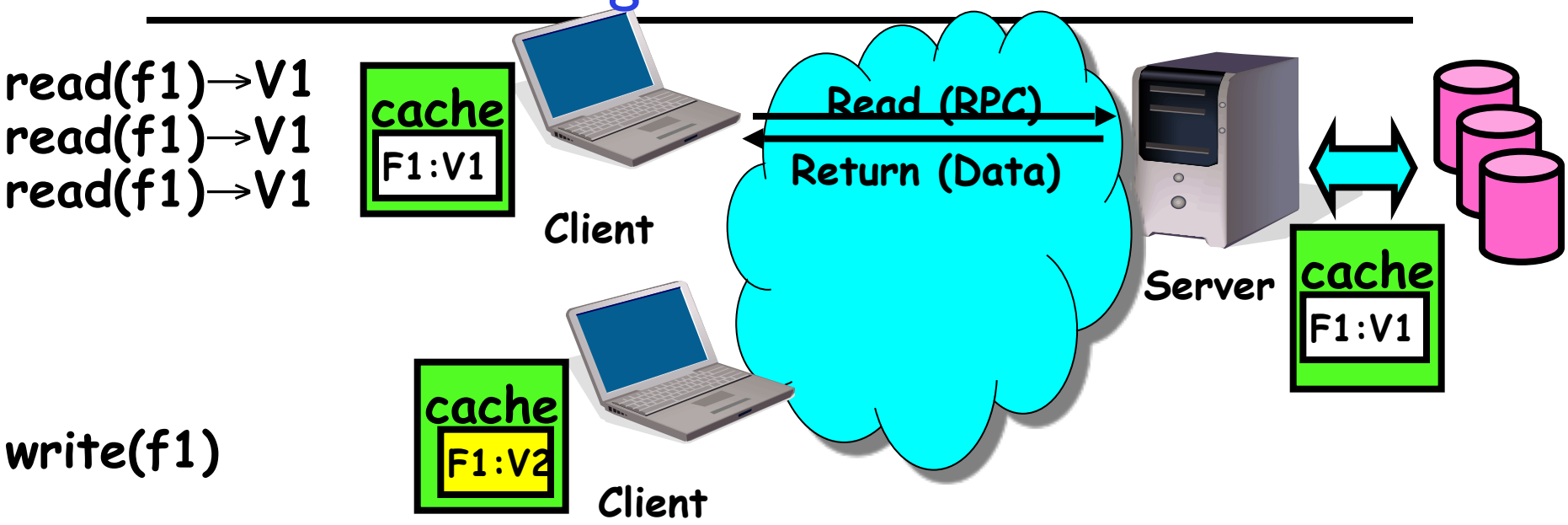
Use of caching to reduce network load



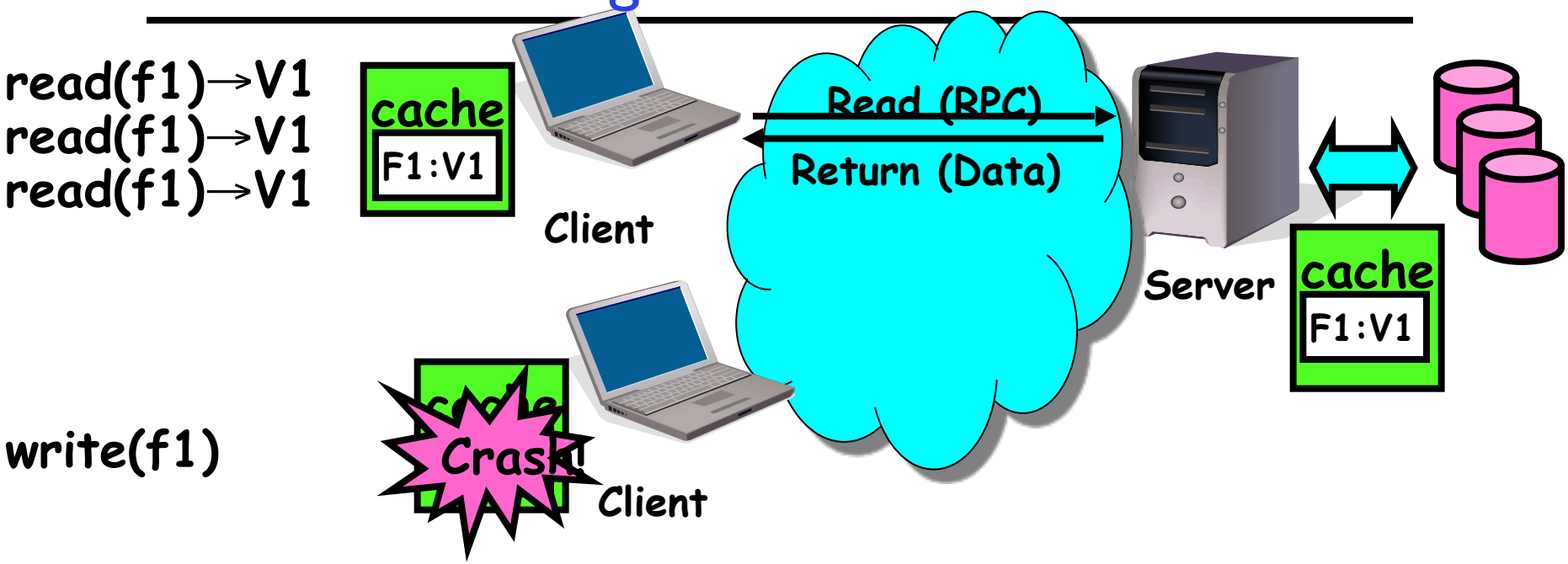
Use of caching to reduce network load



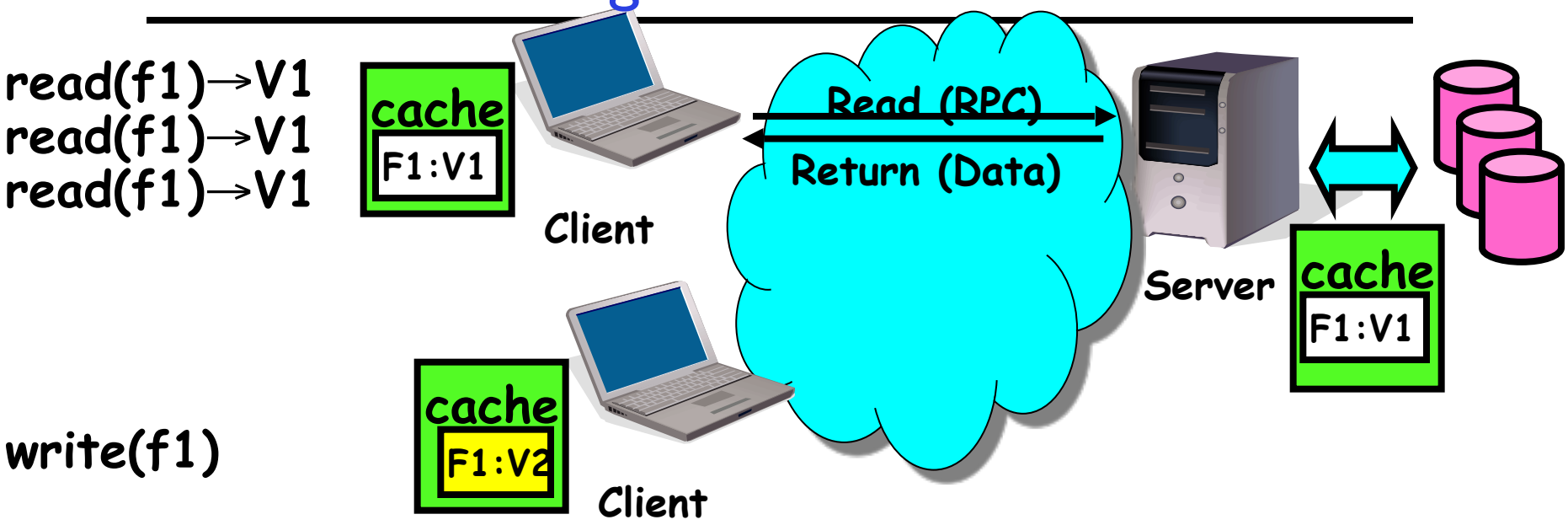
Use of caching to reduce network load



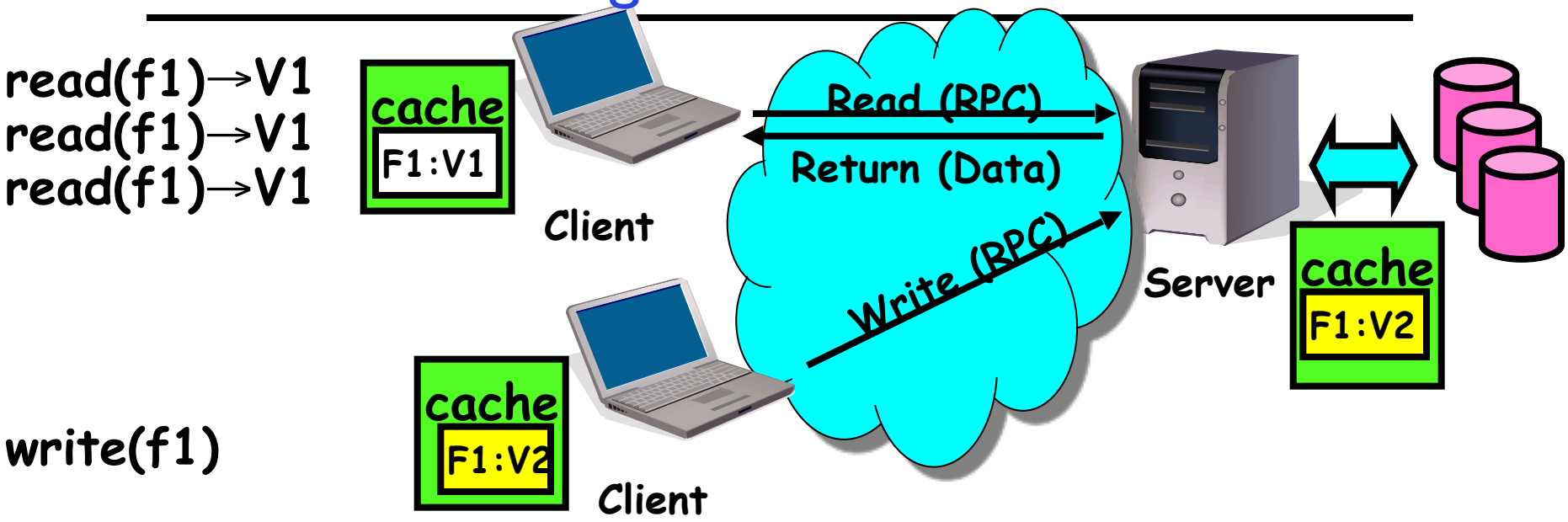
Use of caching to reduce network load



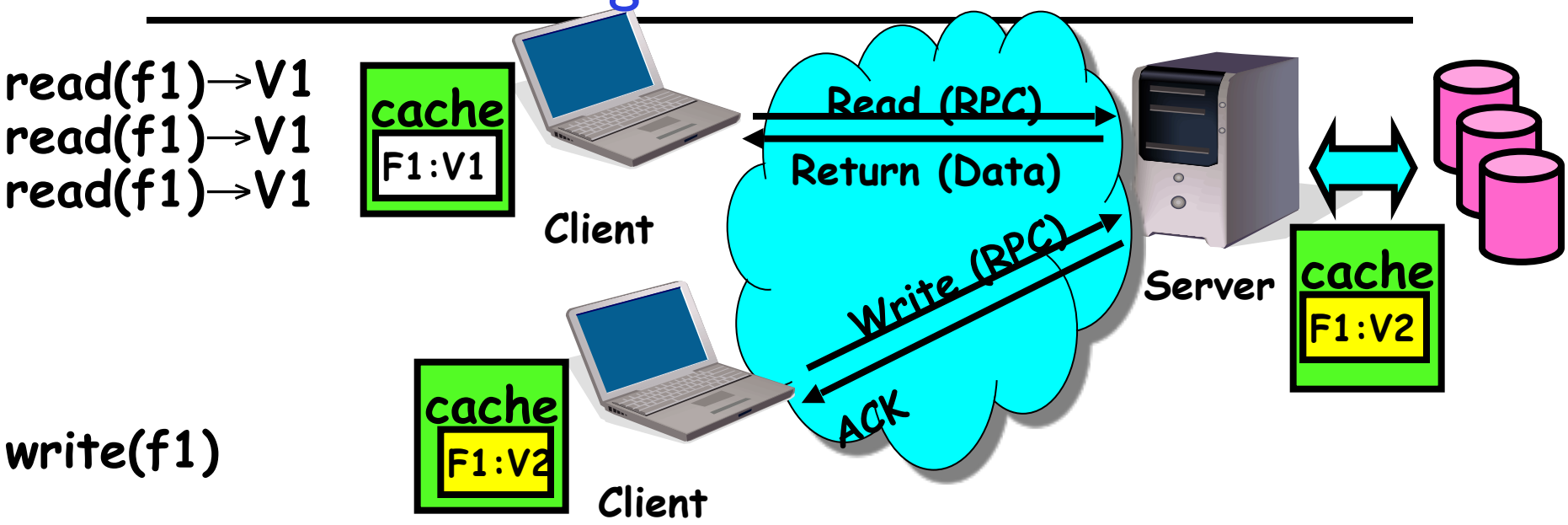
Use of caching to reduce network load



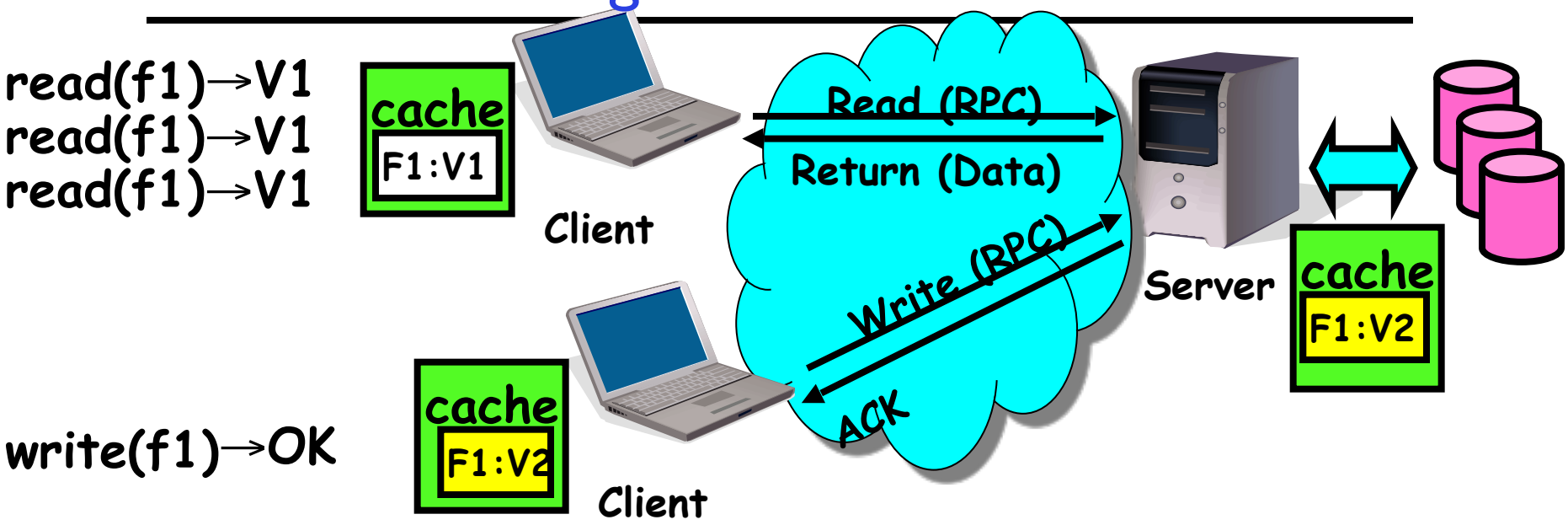
Use of caching to reduce network load



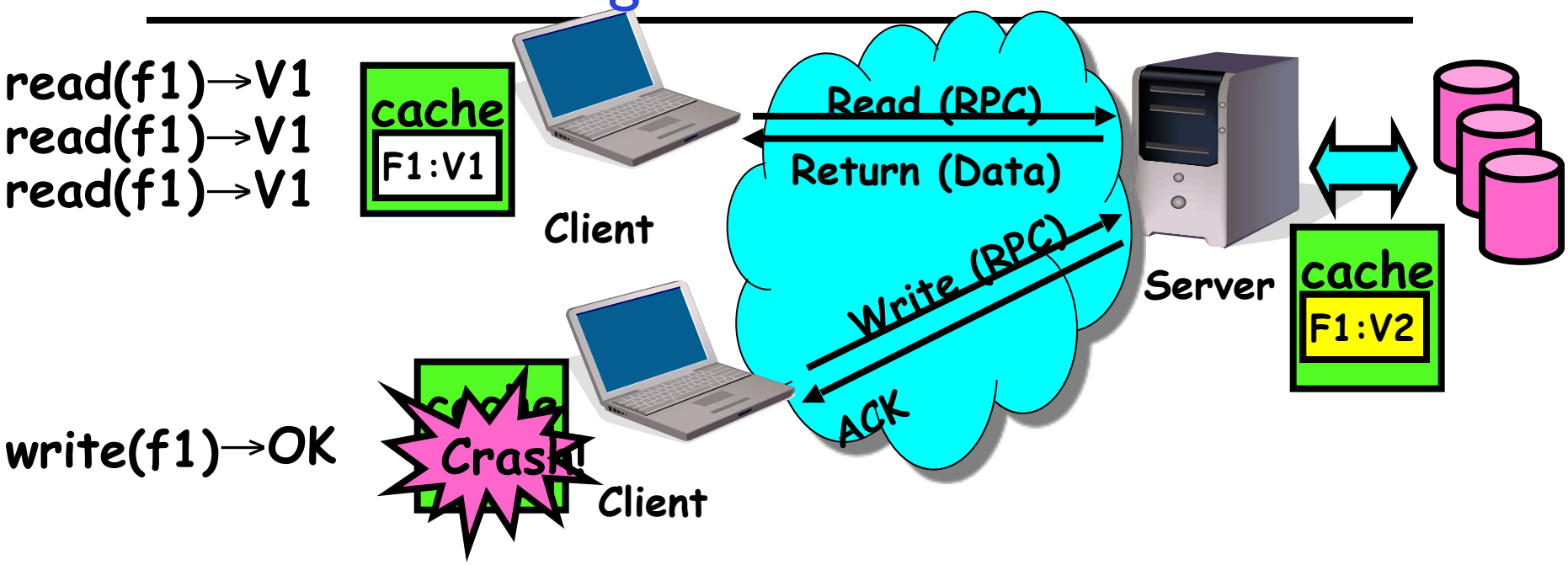
Use of caching to reduce network load



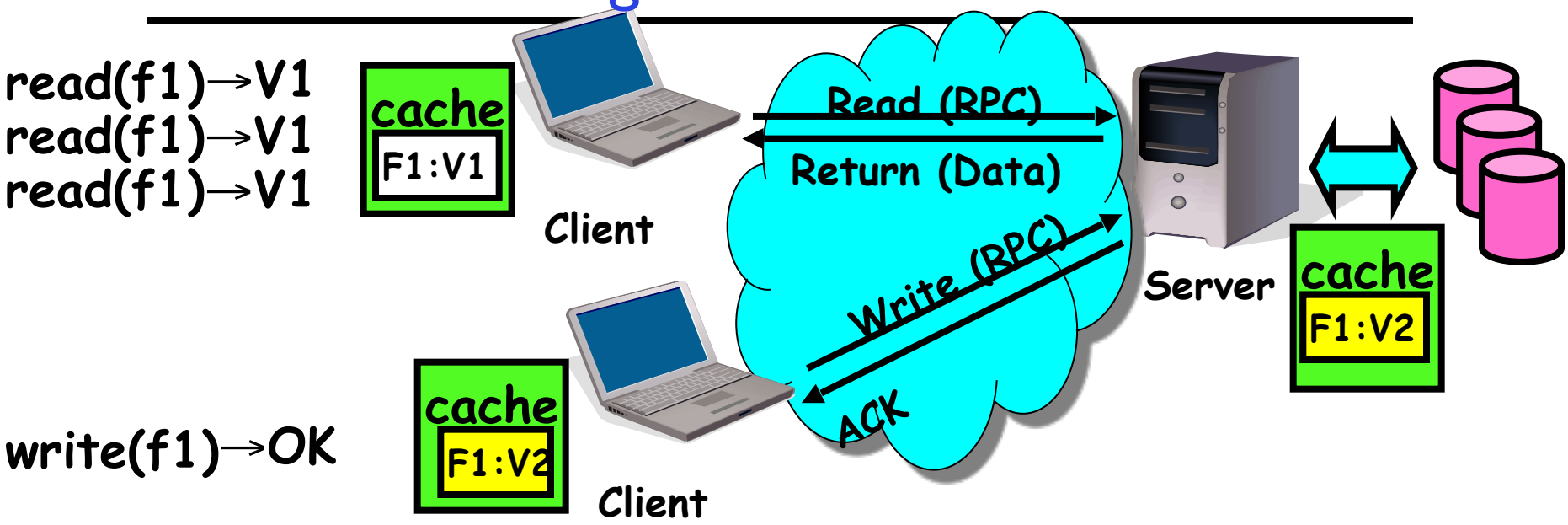
Use of caching to reduce network load



Use of caching to reduce network load



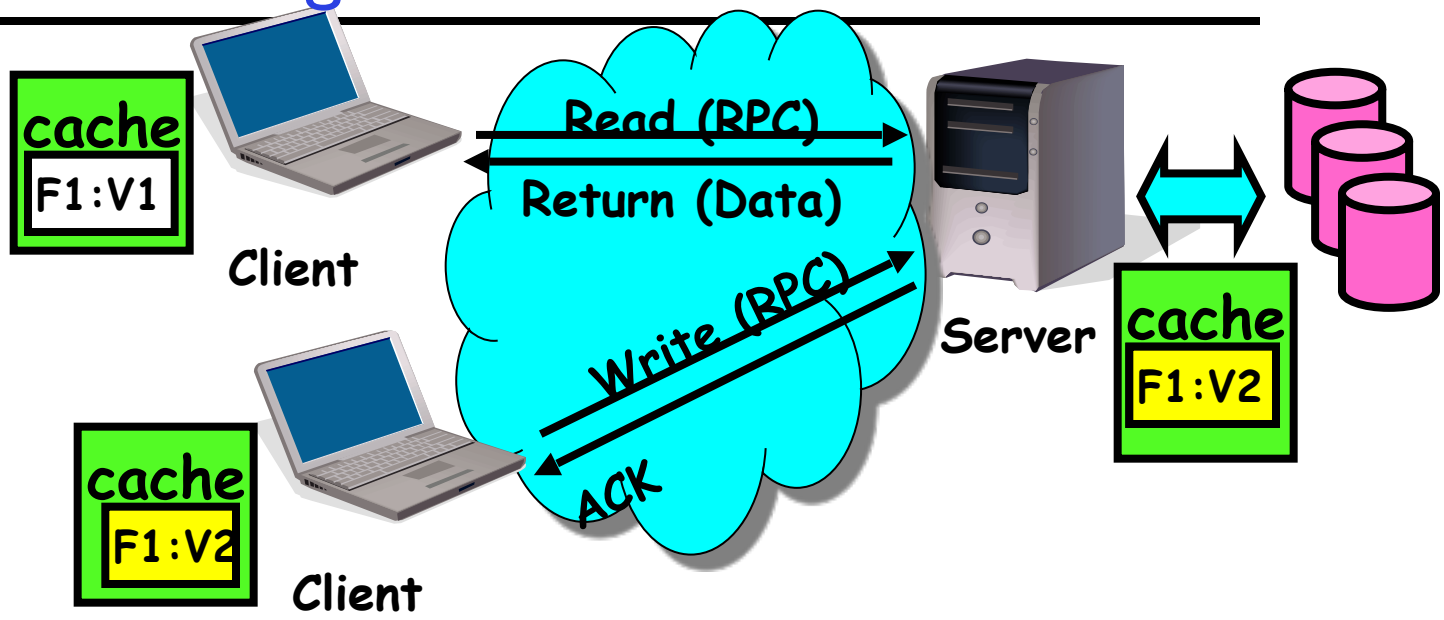
Use of caching to reduce network load



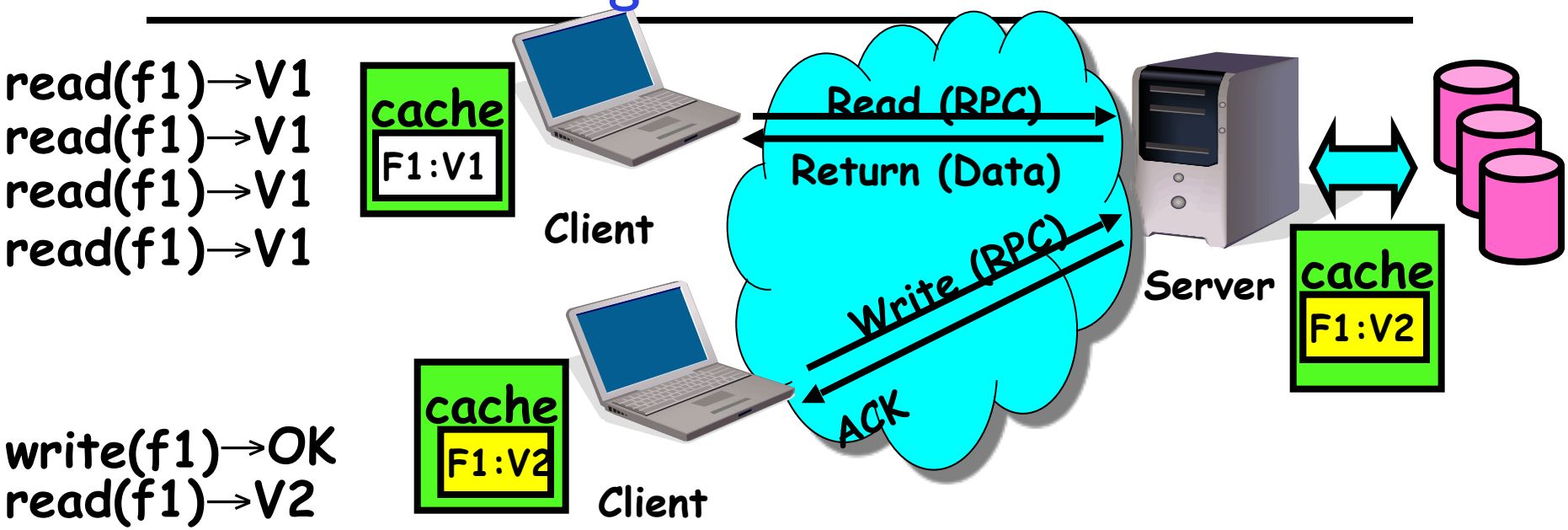
Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V1

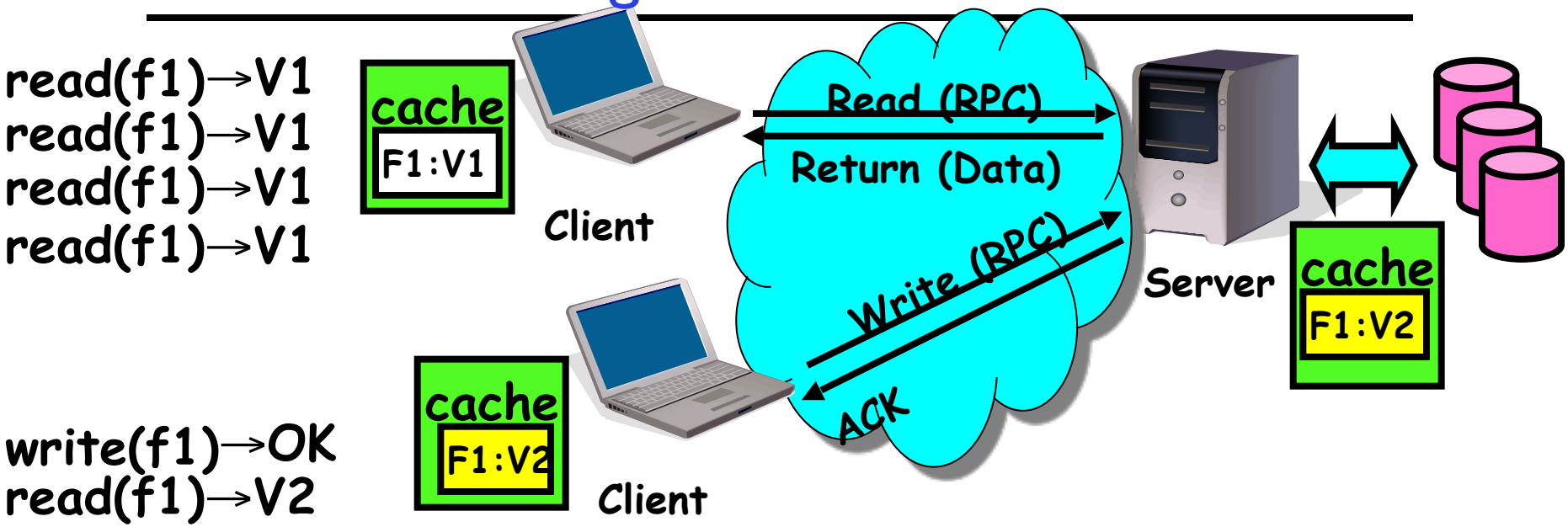
write(f1)→OK



Use of caching to reduce network load



Use of caching to reduce network load



- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Dealing with Failures

- What if server crashes? Can client wait until it comes back and just continue making requests?
  - Changes in server's cache but not in disk are lost

- What if there is shared state across RPC's?
  - Client opens file, then does a seek
  - Server crashes
  - What if client wants to do another read?
- Similar problem: What if client removes a file but server crashes before acknowledgement?

Stateless Protocol

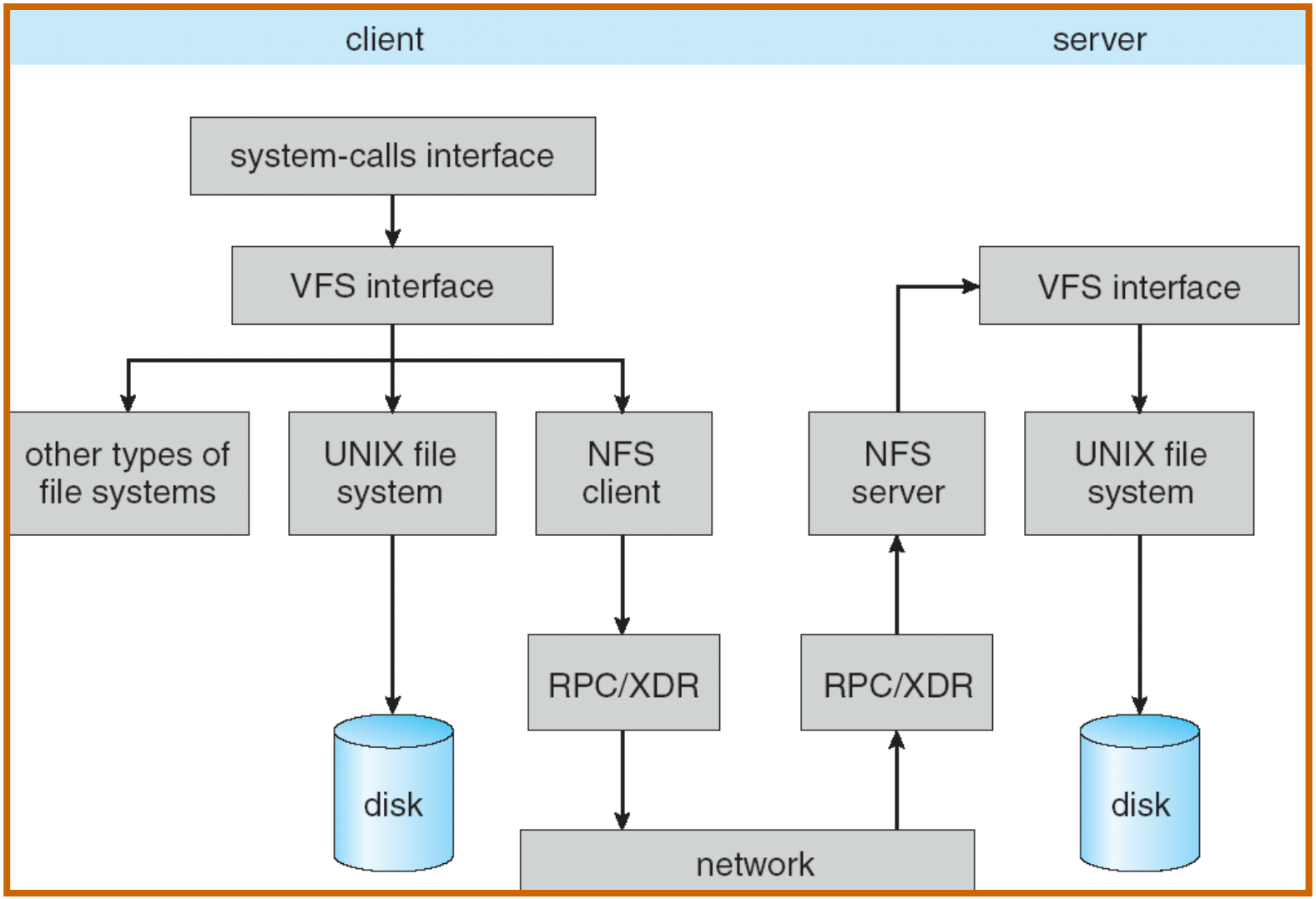
- A protocol in which all information required to service a request is included with the request
- Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)

- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session

Network File System (Sun)

- Defines an RPC protocol for clients to interact with a file server
  - E.g., read/write files, traverse directories, …
  - Stateless to simplify failure cases
- Keeps most operations idempotent
  - Even removing a file: Return advisory error second time
- Don't buffer writes on server side cache
  - Reply with acknowledgement only when modifications reflected on disk

NFS Architecture



Network File System (NFS)

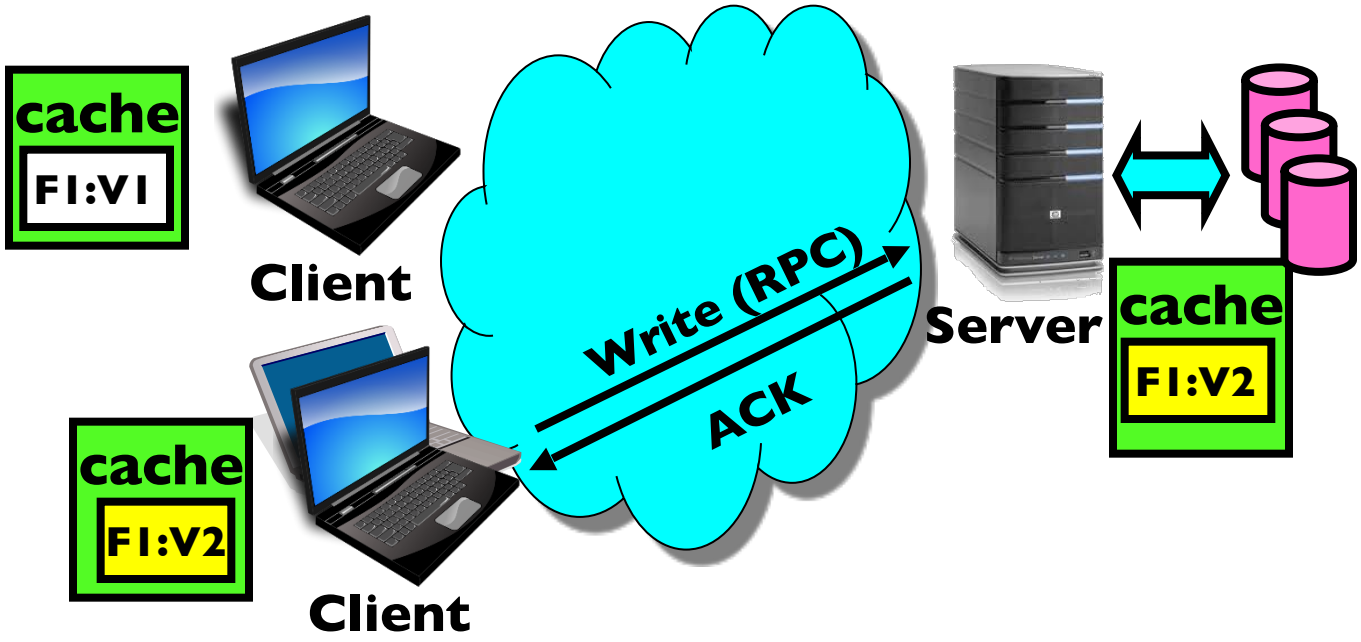
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

NFS Continued

- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - » Hang until server comes back up (next week?)
    - » Return an error. (Of course, most applications don't know they are talking over network)

NFS Cache consistency

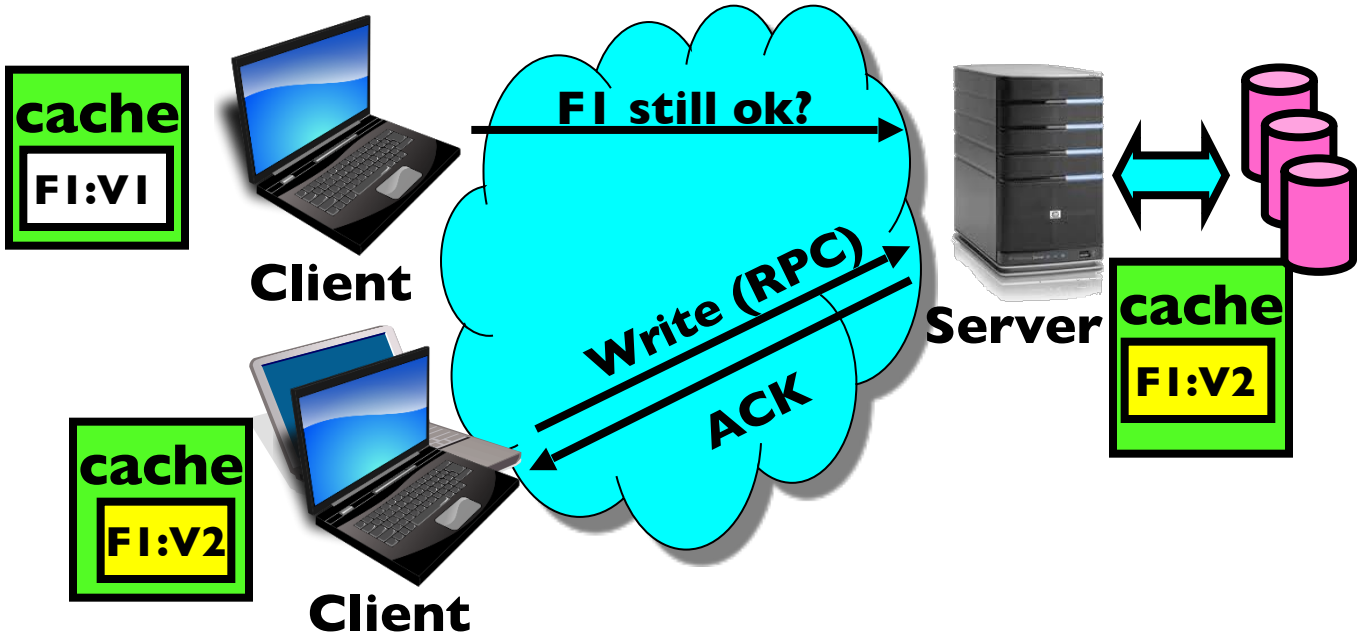
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

NFS Cache consistency

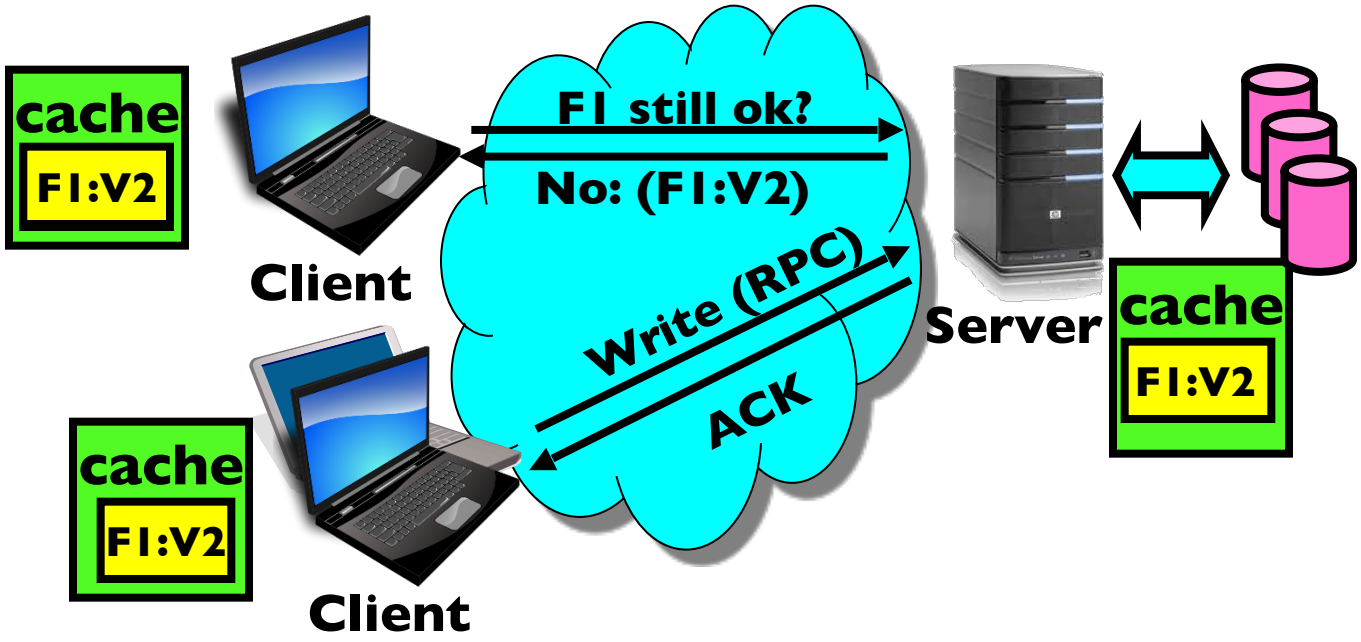
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

NFS Cache consistency

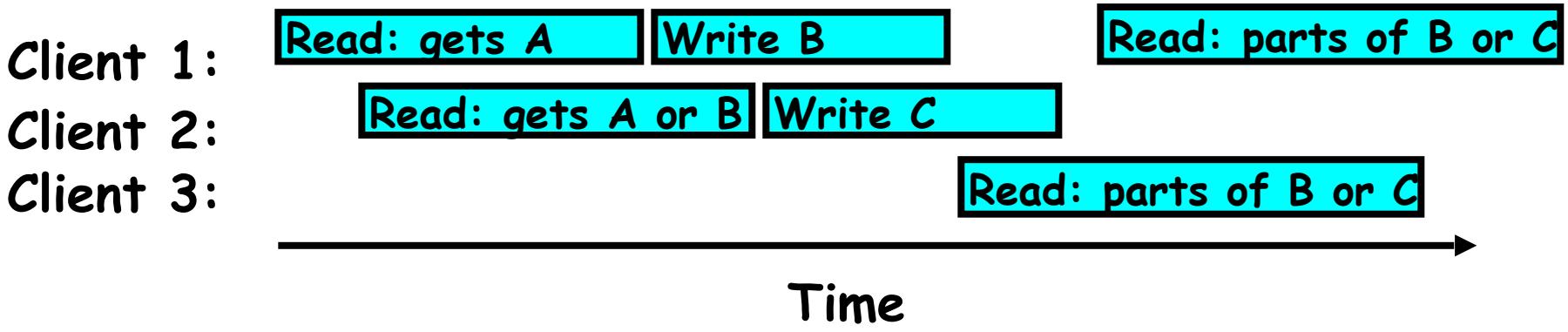
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Summary (1/3)

- 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

Summary (2/3)

- Distributed File System:
  - Transparent access to files stored on a remote disk
  - Caching for performance
- VFS: Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems
- Cache Consistency: Keeping client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
  - NFS: check periodically for changes
  - AFS: clients register callbacks to be notified by server of changes

Summary (3/3)

- Key-Value Store:
  - Two operations
    - » put(key, value)
    - » value = get(key)
  - Challenges
    - » Scalability → serve get()'s in parallel; replicate/cache hot tuples
    - » Fault Tolerance → replication
    - » Consistency → quorum consensus to improve put() performance