# Content distribution problem

- People often distribute popular files from mirrors
- But no place to put a fence!

| Please select a mirror | | | |
|---|---|---|---|
| Host | Location | Continent | Download |
| JAIST 北陸 先端科学技術大学院大学 | Ishikawa, Japan | Asia | 1246 kb |
| BELNET | Brussels, Belgium | Europe | 1246 kb |
| VOXROX WE'RE DEDICATED | New York, New York | North America | 1246 kb |
| easynews usenet made easy | Phoenix, AZ | North America | 1246 kb |
| INTERNAP The Internet Performance Company | Atlanta, GA | North America | 1246 kb |
| ibiblio | Chapel Hill, NC | North America | 1246 kb |

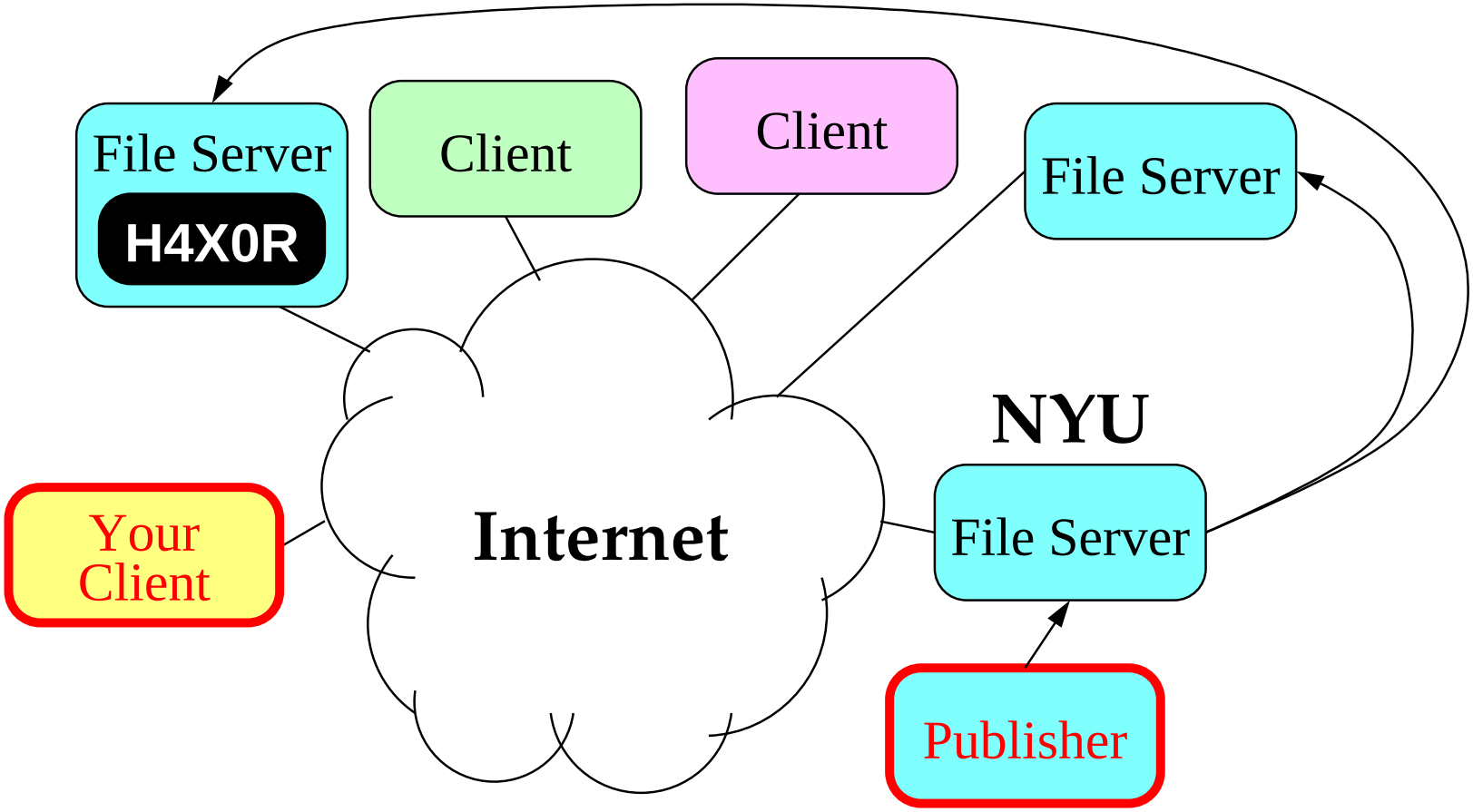
# Signing individual files

- **One solution: Digitally sign files (e.g., w. PGP)**
- **But OS distributions consist of many files:**

```
...                                freetype-2.1.3-6.i386.rpm
cvs-1.11.2-10.i386.rpm             gcc-3.2.2-5.i386.rpm
emacs-21.2-33.i386.rpm             gcc-c++-3.2.2-5.i386.rpm
expat-1.95.5-2.i386.rpm            gdb-5.3post-0.20021129.18.i386.rpm
flex-2.5.4a-29.i386.rpm            glibc-devel-2.3.2-11.9.i386.rpm
fontconfig-2.1-9.i386.rpm          ...
```

- **How do you know file versions go together?**
  - Bad mirror could roll back one file to version with known bug
- **How do you know file name corresponds to contents?**
- **How do you know users will check signature?**

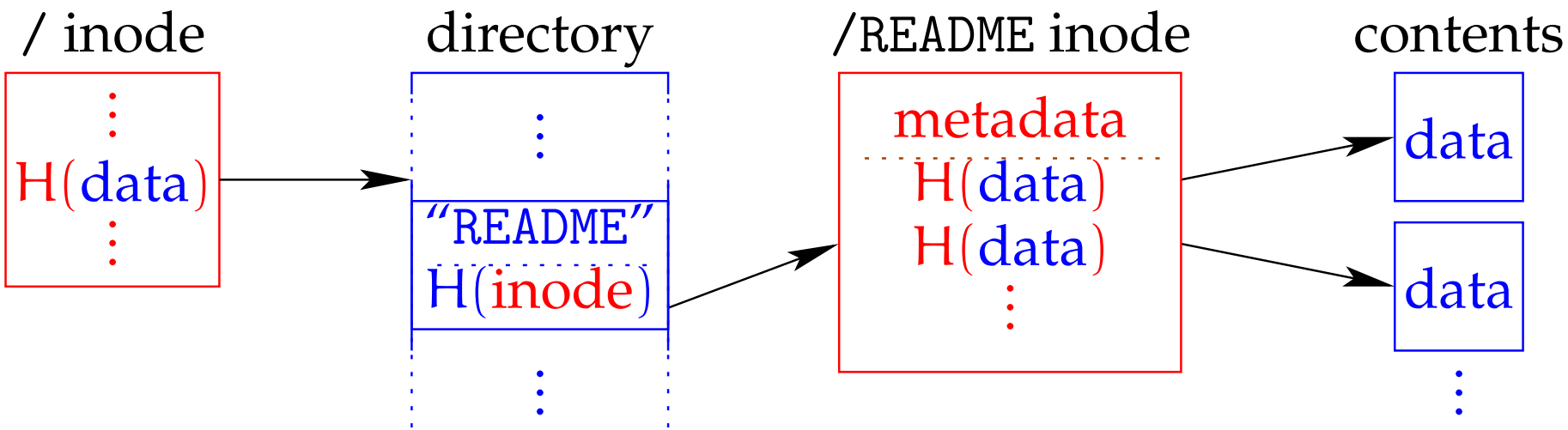
# SFSRO solution: Signing whole file systems




- Give publisher a public signature key
- Tie consistent view of whole FS together with one sig
- Read-only FS interface works with all apps (rpm, . . . )

# Applying Merkle trees to file systems

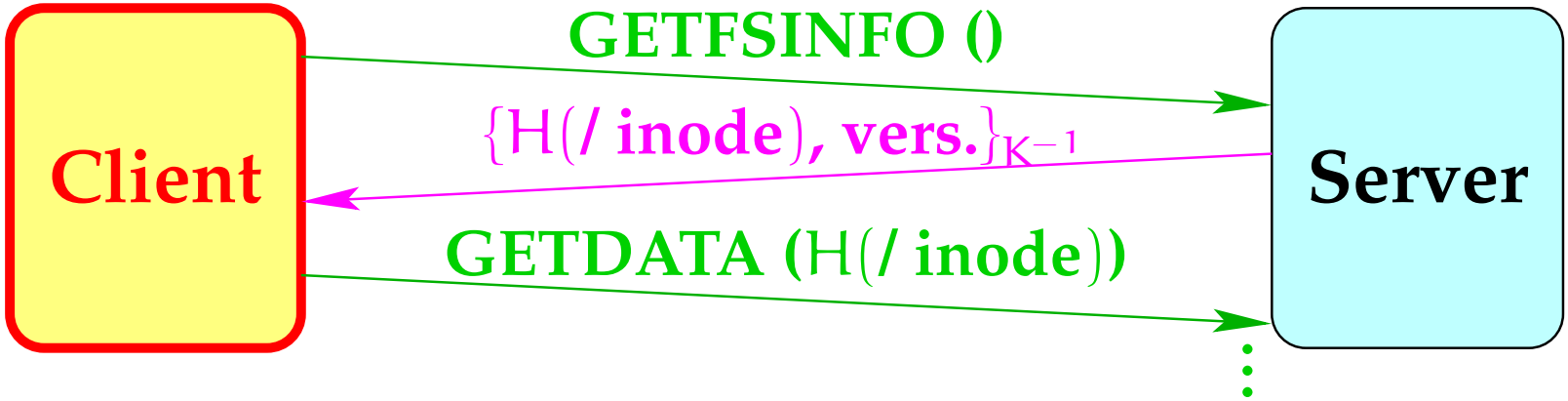
- **Can't just sign raw disk image (too big)**
  - Users may want to download and verify only a few files
- **Idea: Index all data & metadata by cryptographic hash**



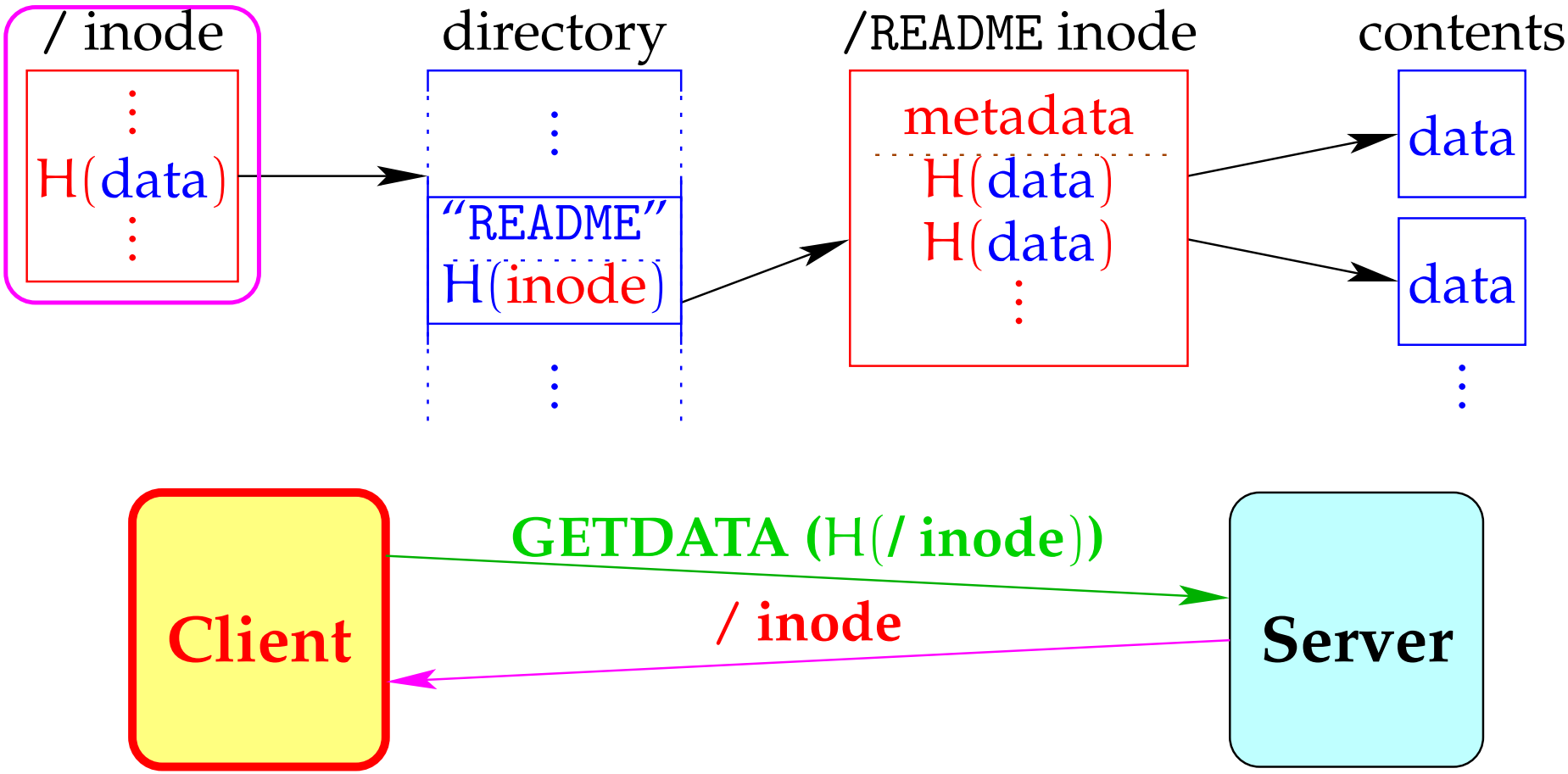

  - H is a fixed-size, collision-resistant hash function
- **Publisher signs hash of root inode**

# SFSRO Protocol

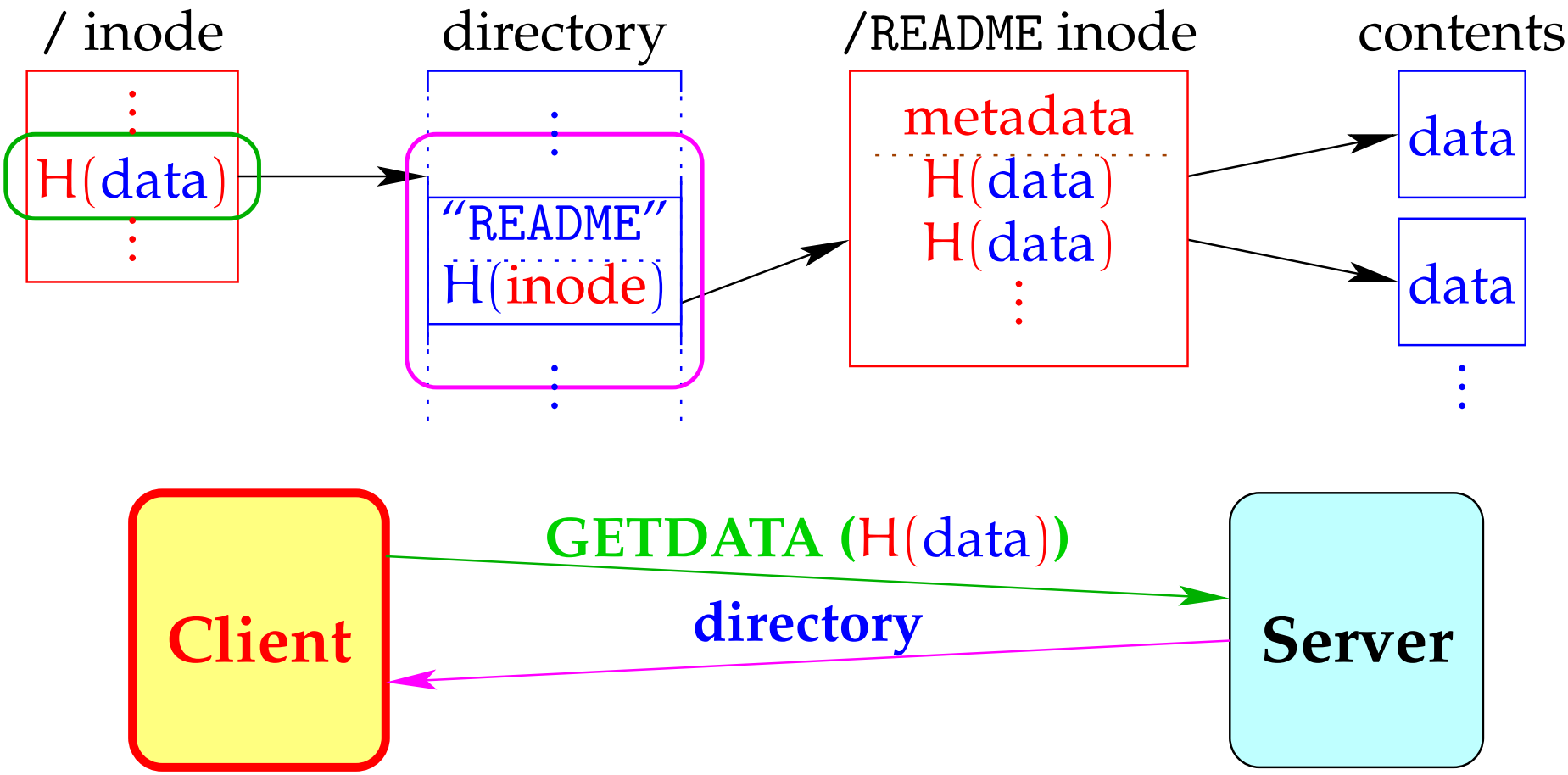
- **GETFSINFO () – Get signed hash of root directory**
- **GETDATA (*hash*) – Get block with *hash* value**
- **Example: To read file** `/sfs/@host,cz···a3/README`
  - First get signed hash, then walk down tree



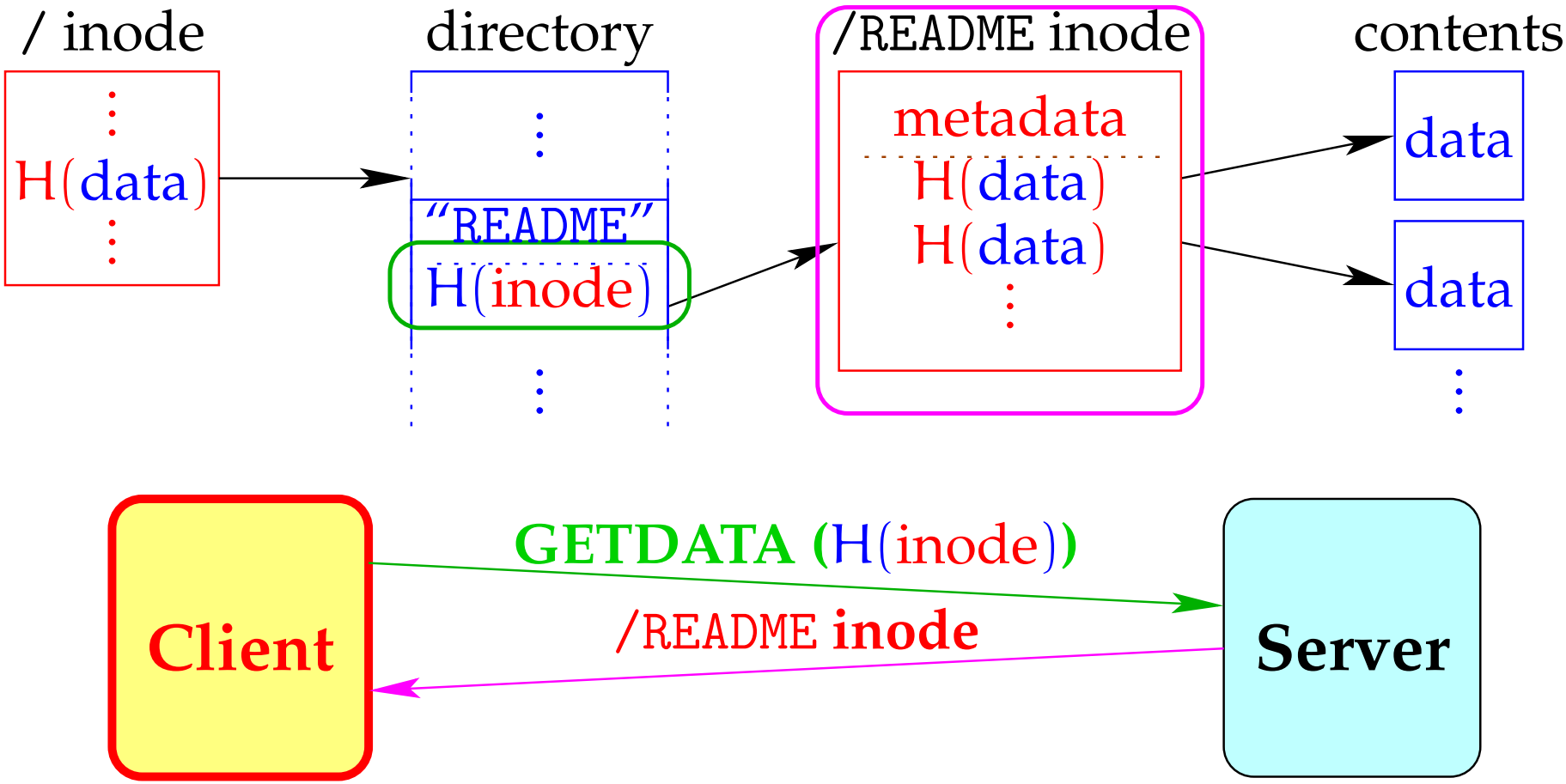
# SFSRO Protocol



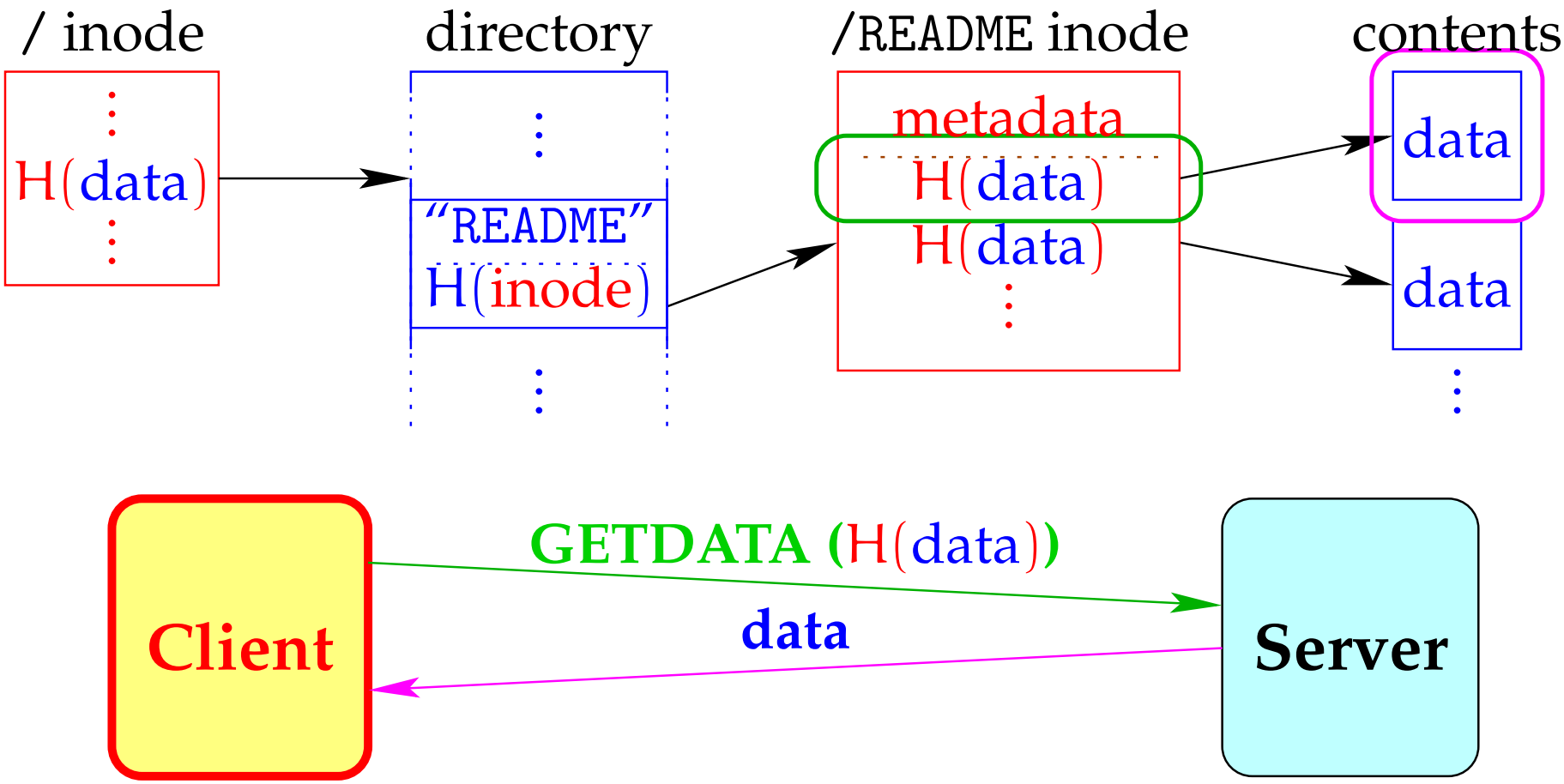
# SFSRO Protocol



# SFSRO Protocol

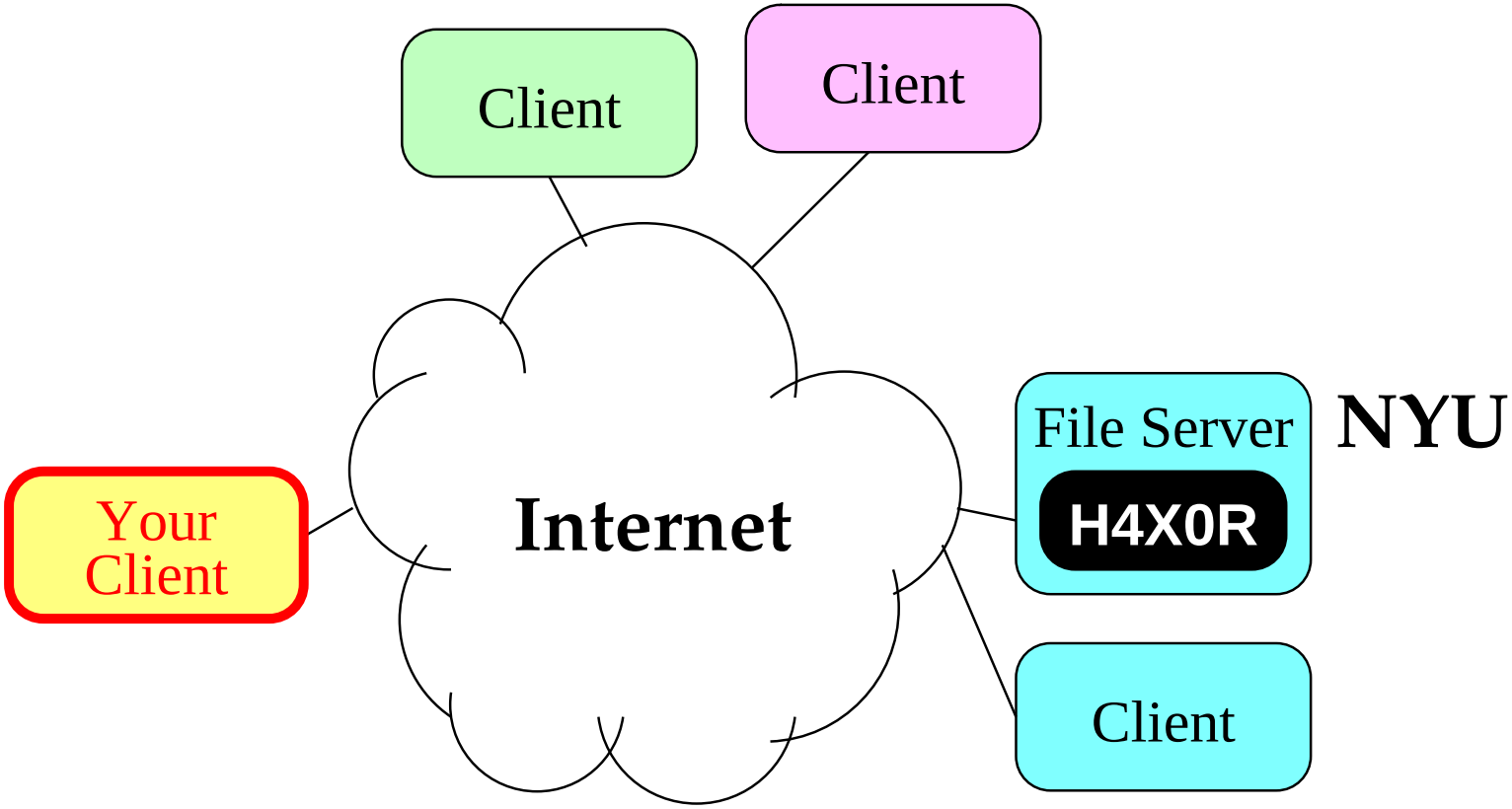


# SFSRO Protocol



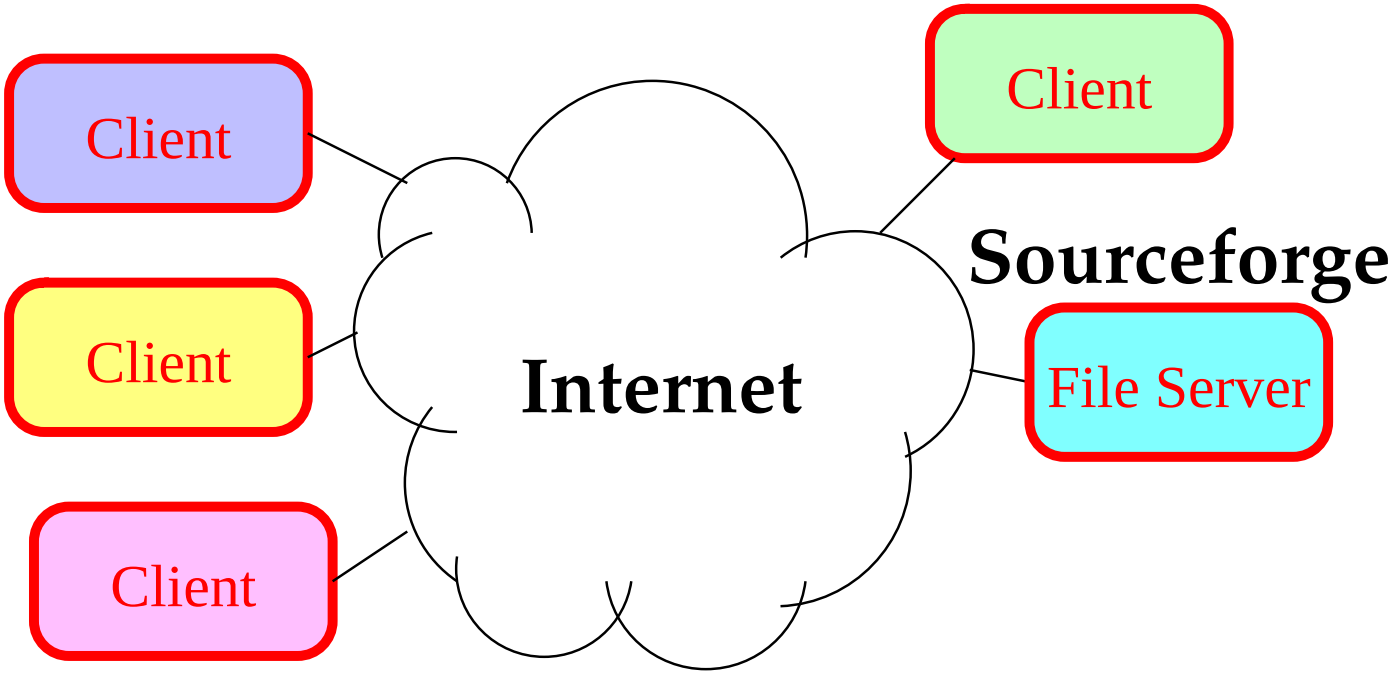
# Read-Write File Systems

# SUNDR: Putting the server outside the fence




- **Normally trust file servers to return correct data**
- **…and to reject unauthorized requests**
- ***Don't* need to trust SUNDR server**
  - Can detect misbehavior even if attacker controls server

# Motivation: Outsourcing data storage

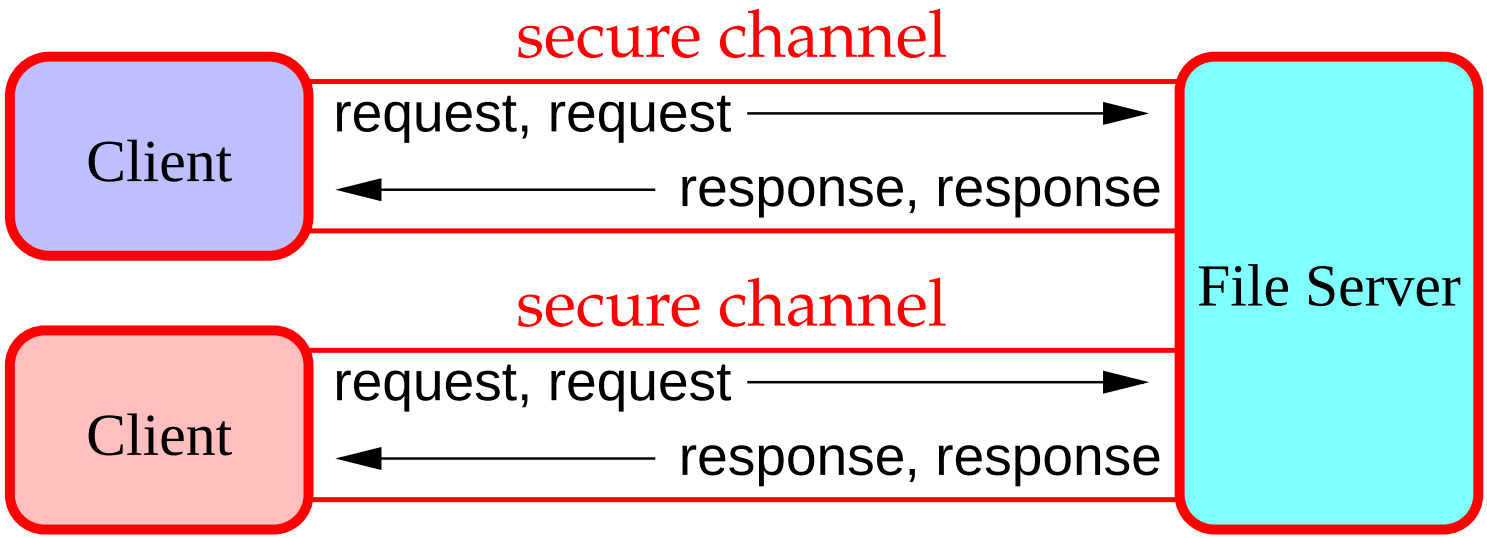



- E.g., Sourceforge hosting source repositories
- Attractive target of attack

# A worrisome trend

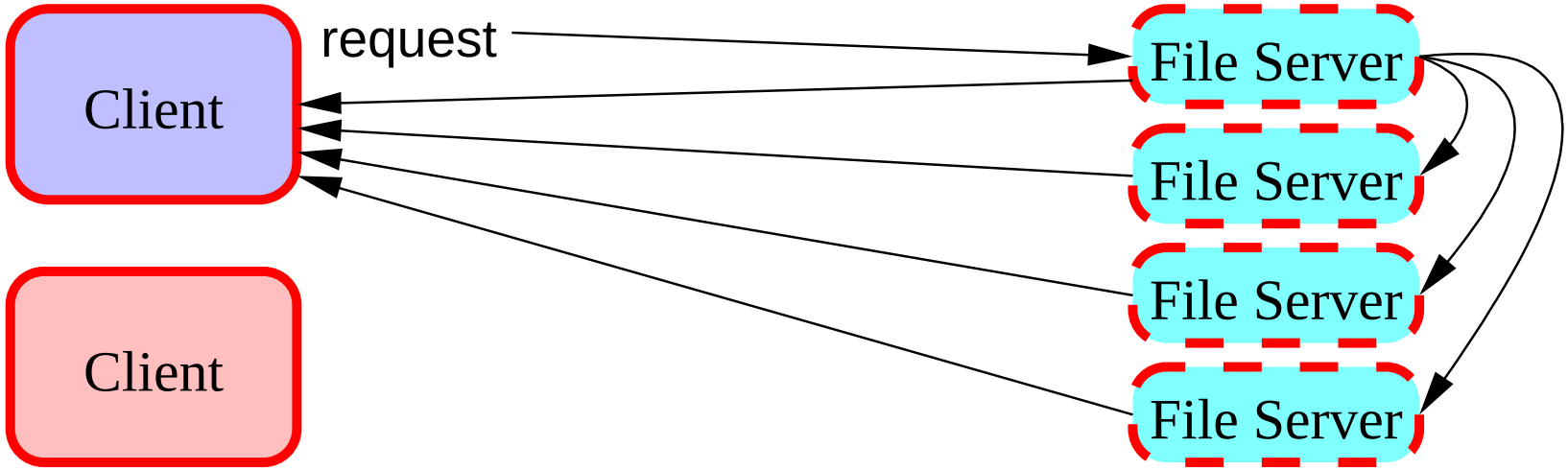
- **5/17/01: Apache development servers compromised**
  - Password captured by trojaned ssh binary at sourceforge
  - The integrity of all source code repositories is being individually verified by developers... - Apache press release
- **11/20/03: Debian administrators discover "root kit"**
  - at the time the break-ins were discovered... it wasn't possible to hold [the release] back anymore. – Debian report
- **3/23/04: Gnome server compromise discovered**
  - We think that the released gnome sources and the ... repository are unaffected.... we are cautiously hopeful that the compromise was limited in scope. – Owen Taylor

# Traditional file system model



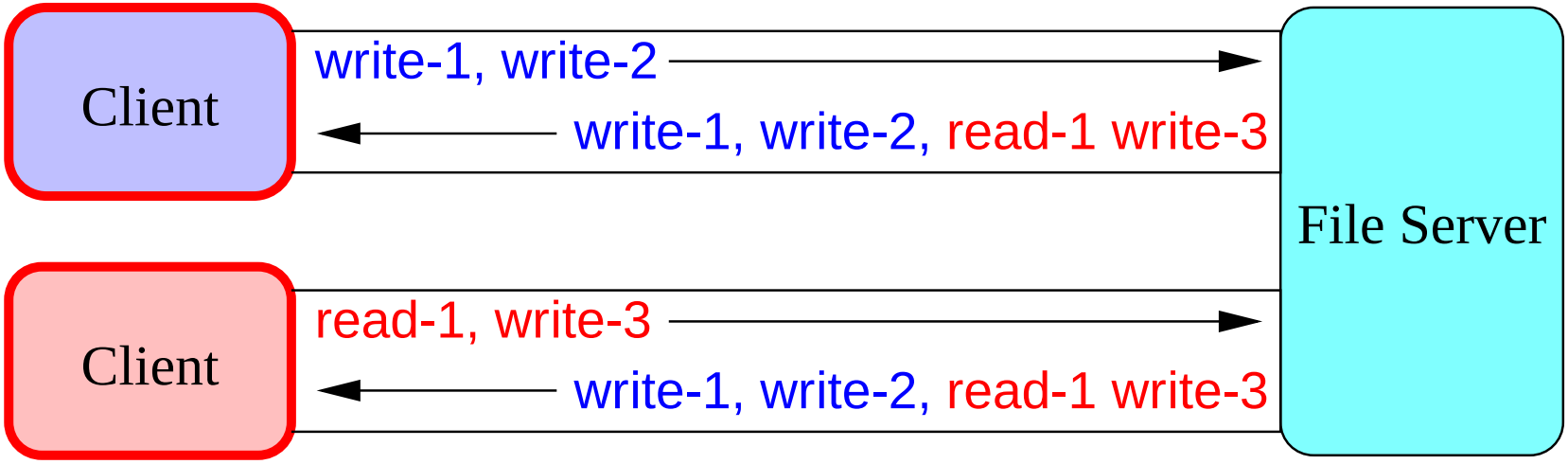

- **Clients & servers communicate over secure channels**
  - Network attackers can't tamper with requests
- **Server can't prove what requests it received**
  - Trust server to execute requests properly
  - Trust server to return correct responses

# BFS model



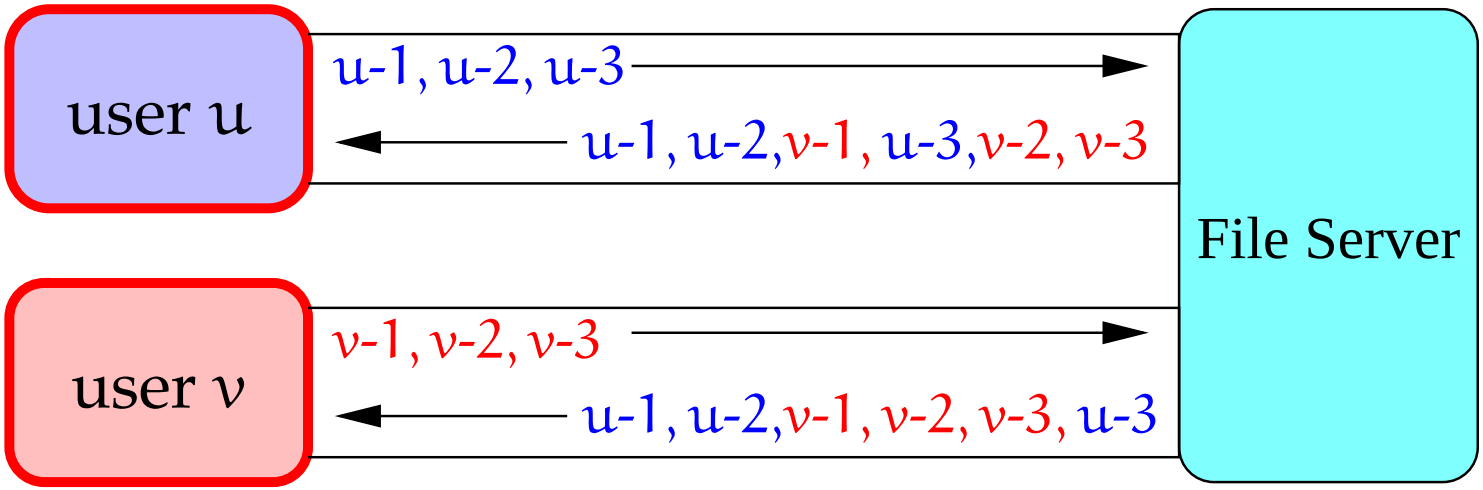

- **Replicate server 4 times**
  - Client gets response from 4 servers
  - Require 2 servers to agree that 3 servers agree on response
  - Means okay if one server controlled by attacker

# SUNDR model




- **Clients send digitally signed requests to server**
  - This is now possible with sub-millisecond digital signatures
- **Server does not execute anything**
  - Just stores signed requests from clients
  - Answers a request with other signed requests, proving result
  - Does not know signing keys—cannot forge requests

# Danger: Dropping & re-ordering




- **Server can drop signed requests**
  - E.g., back out critical security fix
- **Or show requests to clients in different order**
  - E.g., overwrite new file with old version
  - Can be effectively same as dropping requests

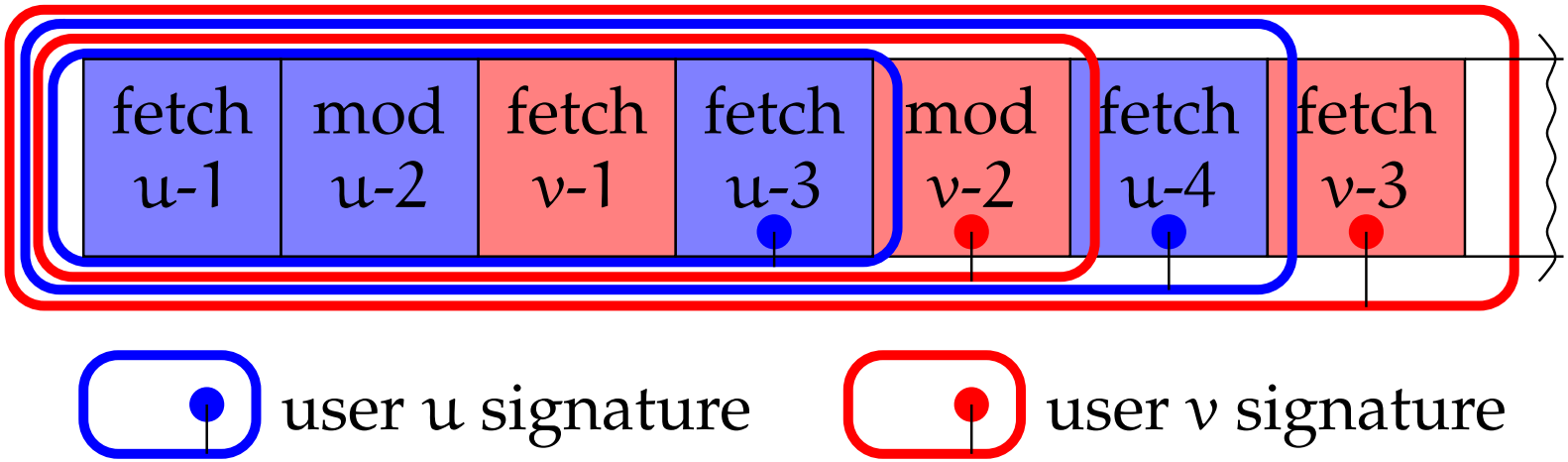
# A Fetch-Modify interface

- **Need to specify FS correctness condition**
  - Many file system requests in POSIX
  - Far too complex to formalize
- **Boil FS interface down to two request types:**
  - *Fetch* – Client validates cached file or downloads new data
  - *Modify* – One client makes new file data visible to others
  - Can map system calls onto fetch & modify operations:
    open $\rightarrow$ fetch (dir & file), write+close $\rightarrow$ modify,
    truncate $\rightarrow$ modify, creat $\rightarrow$ fetch+modify, ...

# File system correctness

- **Goal:** ***fetch-modify consistency***
  - System orders operations reasonably [linearizability]
  - A fetch reflects exactly the authorized modifications that happened before it
  - (Sometimes called "close-to-open consistency")
- **How close can we get with an untrusted server?**
  - A: *Fork consistency*
- **Next: 3 progressively more realistic realizations**
  - Signed logs (enormous bandwidth & FS-wide lock)
  - Serialized SUNDR (FS-wide lock)
  - SUNDR

# Solution 1: Signed logs



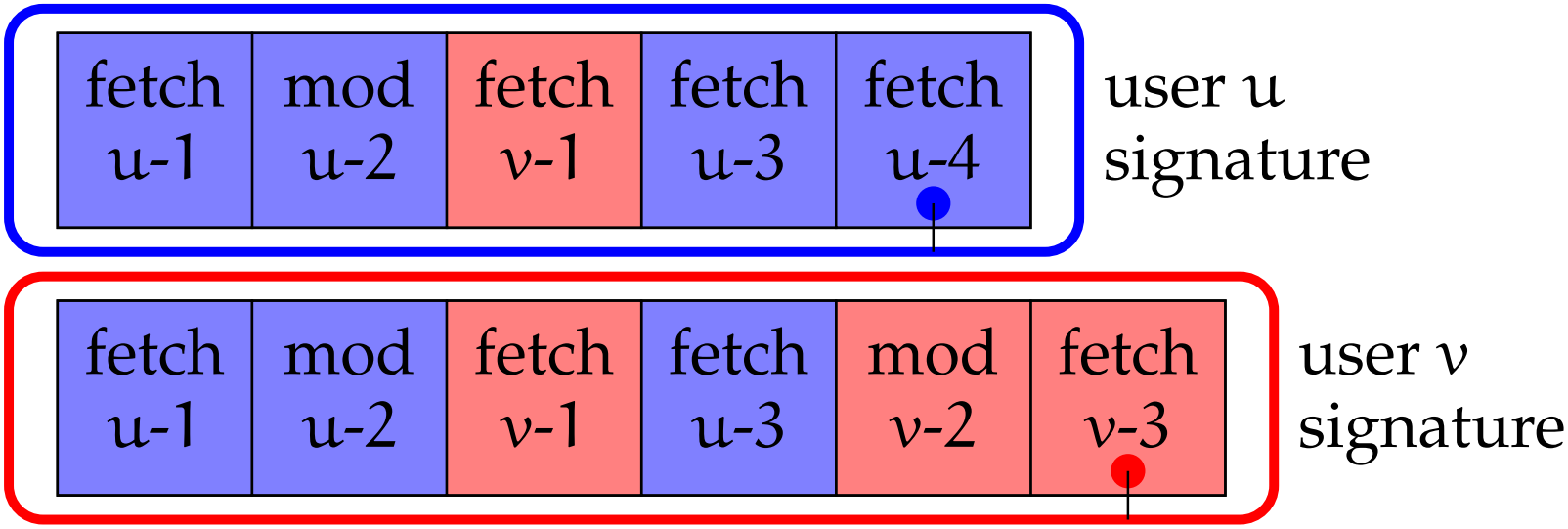

- **Detect reordering by signing entire FS history:**
- PREPARE **RPC – lock file system, download log**
  - Client checks signatures on log entries
  - Client checks that its previous operation is still in log
- **Client plays log to reconstruct FS state**
- **Client appends new operation, signs new log**
- COMMIT **RPC – upload signed log, release lock**

# Signed log security properties

- **Server cannot manufacture operations**
  - Clients check signatures, which server can't forge
- **Server cannot undo operations already revealed**
  - Clients check their last operation is in current log
- **Server cannot re-order signed operations**
  - Signatures over past history would become invalid

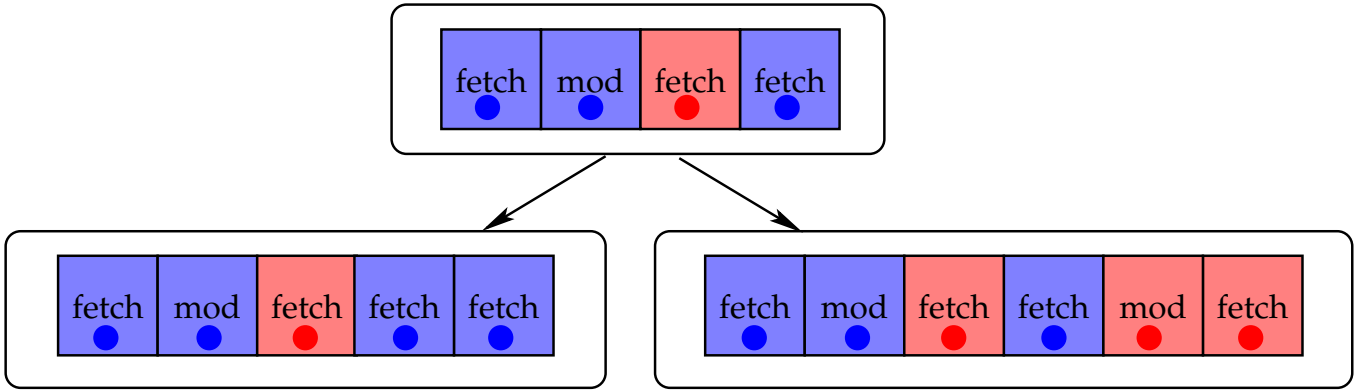
# What can a malicious server do?

- **Server can mount a *fork attack***
  - Conceal clients' operations from one another
  - But produces divergent logs for different users
- **Suppose server doesn't lock, conceals mod *v*-2 from *u***




  - Either client can detect given any later log of the other

# Fork consistency




- **User's views of file system may be forked**
  - But operations in each branch fetch-modify consistent
  - Can't undetectably re-join forked users
- **Best possible consistency w/o on-line trusted party**
  - Say $u$ logs in, modifies file, logs out
  - $v$ logs in but doesn't see $u$'s change
  - No defense against this attack (w/o on-line trusted party)
  - This is the only possible attack on a fork-consistent system

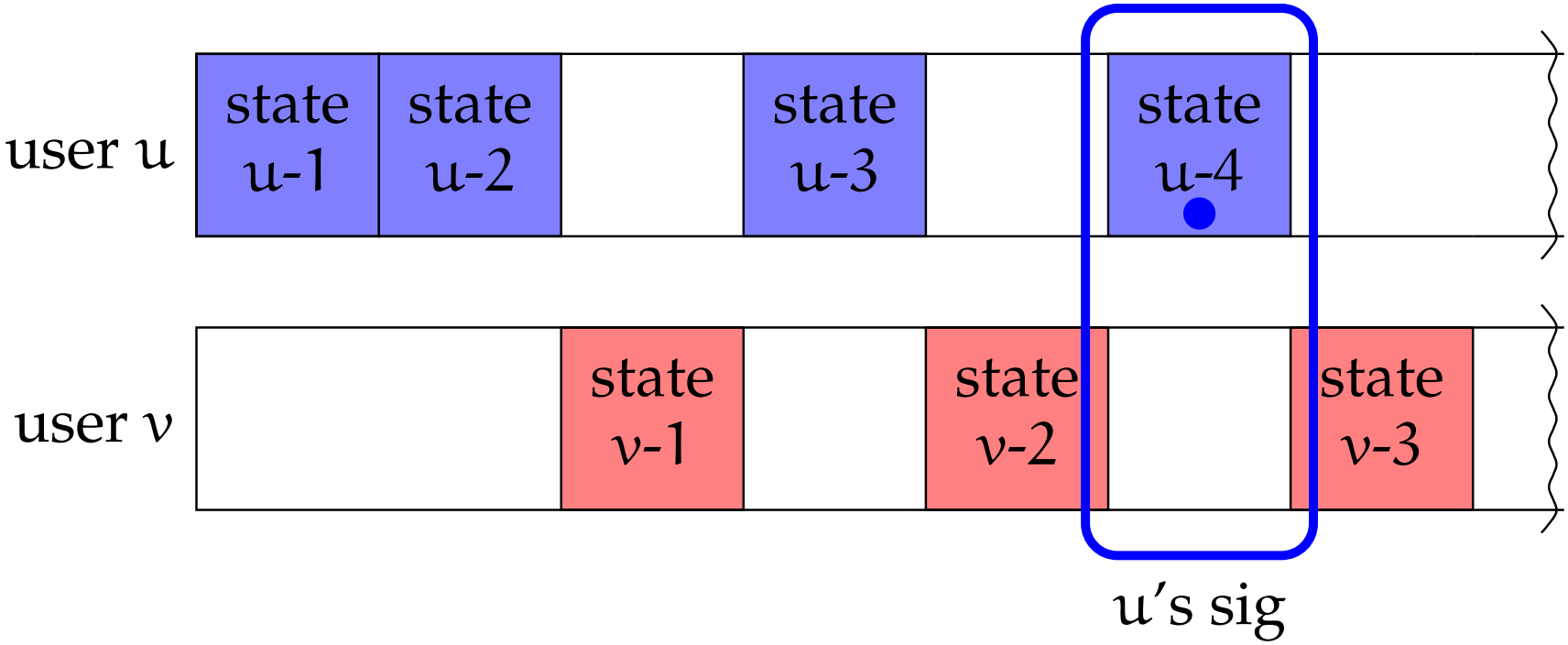
# Implications of fork consistency

- **Can trivially audit server retroactively**
  - If you see operation u-n, you were consistent with u (and transitively anything u saw) at least until u performed u-n
- **Exploit any on-line [semi-]trusted parties to improve consistency**
  - Clients that communicate get fetch-modify consistency
    E.g., two clients on an Ethernet when server "outsourced"
  - Pre-arrange for "timestamp" box to update FS every minute
- **How to recover from a forking attack?**
  - This is actually a well-studied problem!
  - Ficus, CODA reconcile conflicts after net partition
  - Experience: a fork is annoying, but not tragic

# Limitations of signed logs

- **Signed logs achieve fork consistency…**
- **But signed log scheme hopelessly inefficient**
  - Each client must download every operation
  - Each client must reconstruct entire file system state
  - Global lock on file system adds unacceptable overhead
- **Systems with logs typically use checkpoints…**
  - Can we sign SFSRO-like snapshots instead of history?

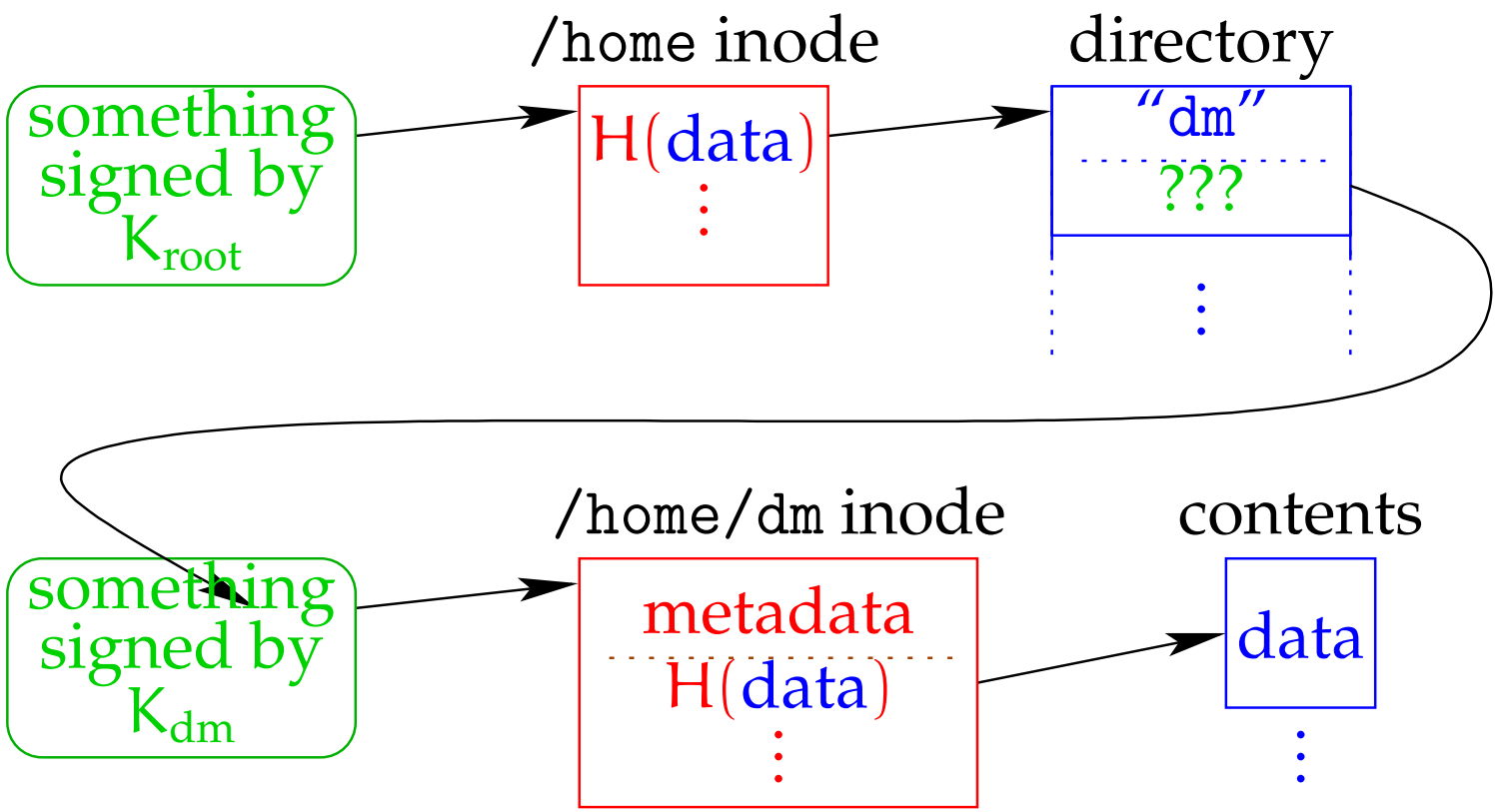
# A plan for signing snapshots



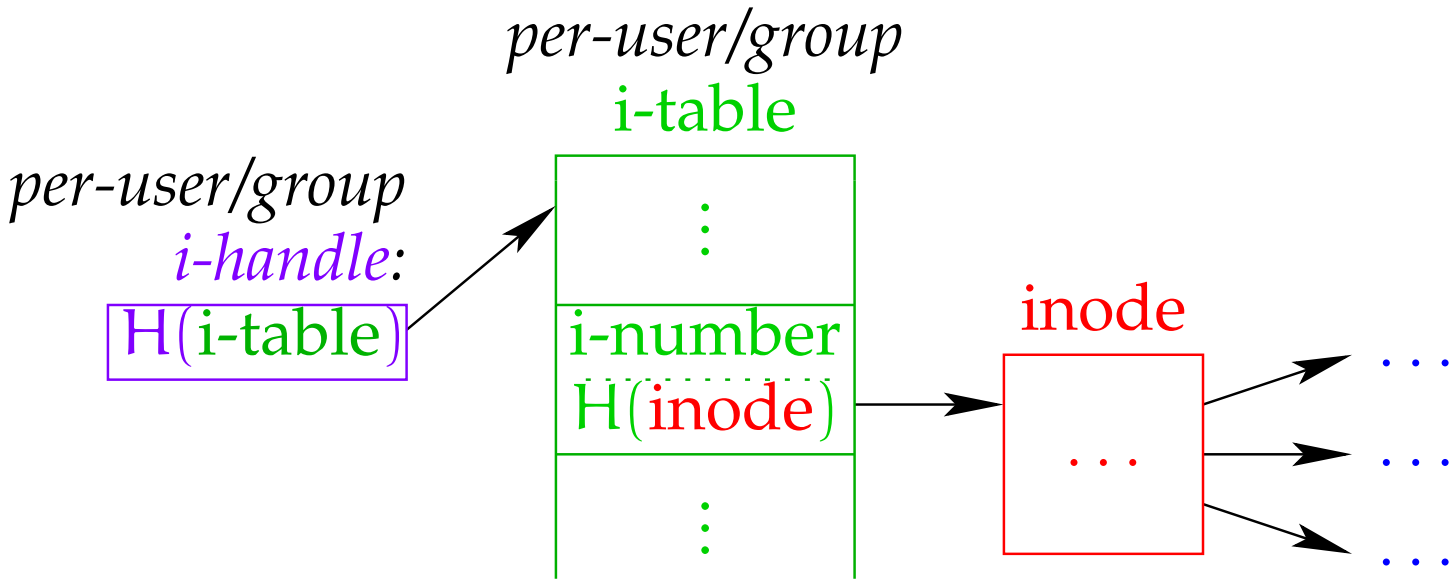

- Somehow represent snapshots of each user's files in a way that they can be combined…
- Somehow prevent re-ordering of users' snapshots…

# Combining snapshots

- **A user's directory might contain another user's file**
  - E.g., "root" owns `/home`, dm owns `/home/dm`
  - Root must sign file name while dm signs contents

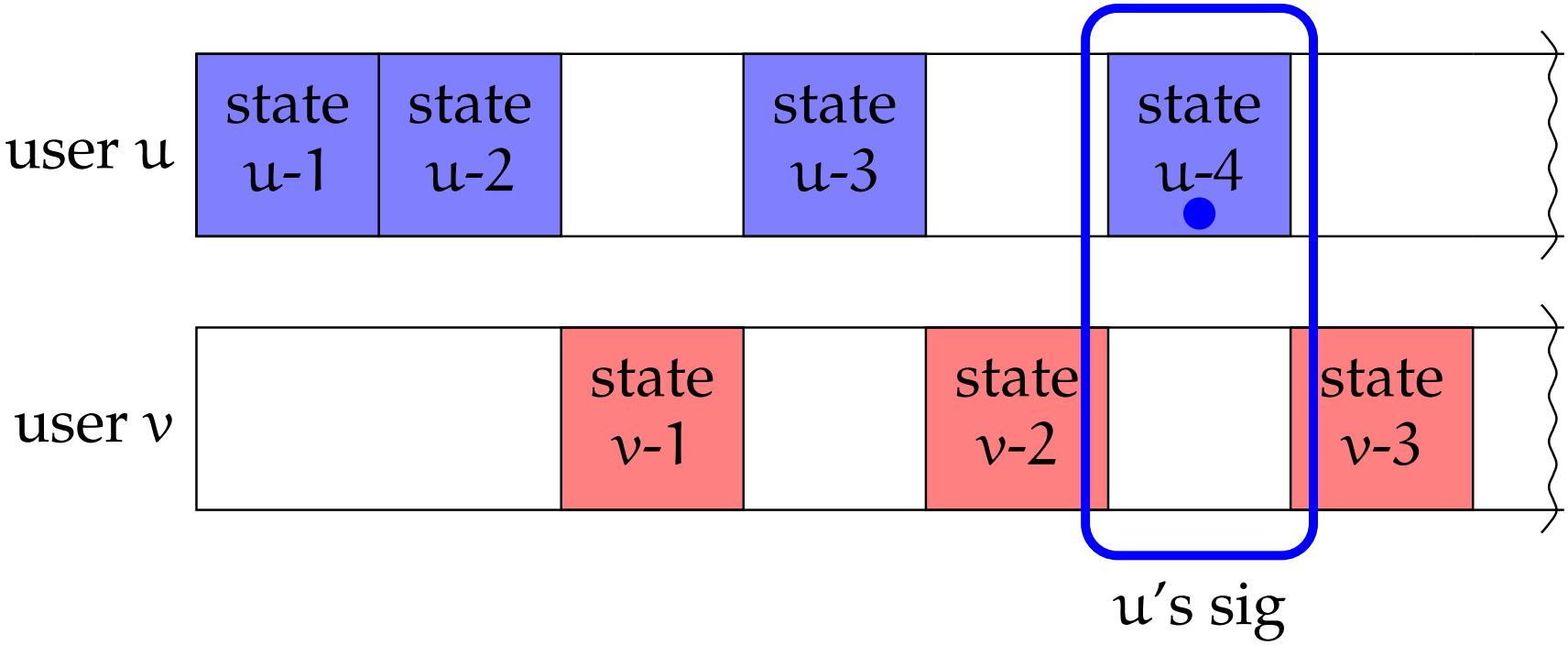


# Per-user or -group i-numbers



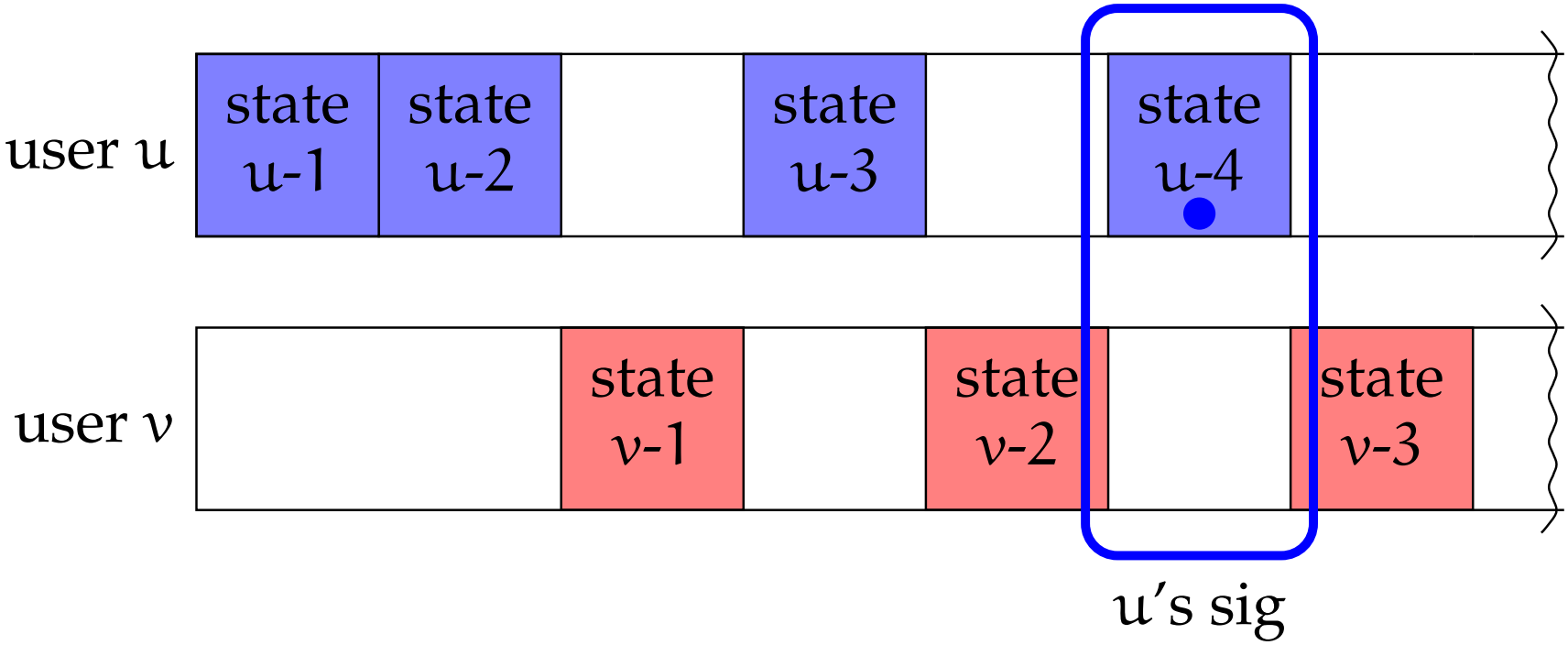

- **Add a level of indirection to SFSRO data structures**
- **SUNDR directory entry:** file name ⟨user/group, i-number⟩
- **Per-user/group *i-tables* map** i-number $\rightarrow$ H(inode)
- **Hash each i-table to a short *i-handle* users can sign**

# A plan for signing snapshots




- ✓ Somehow represent snapshots of each user's files in a way that they can be combined…
- ☐ Somehow prevent re-ordering of users' snapshots…

# Detect re-ordering with version vectors




- **Sign latest version # of every user & group:**

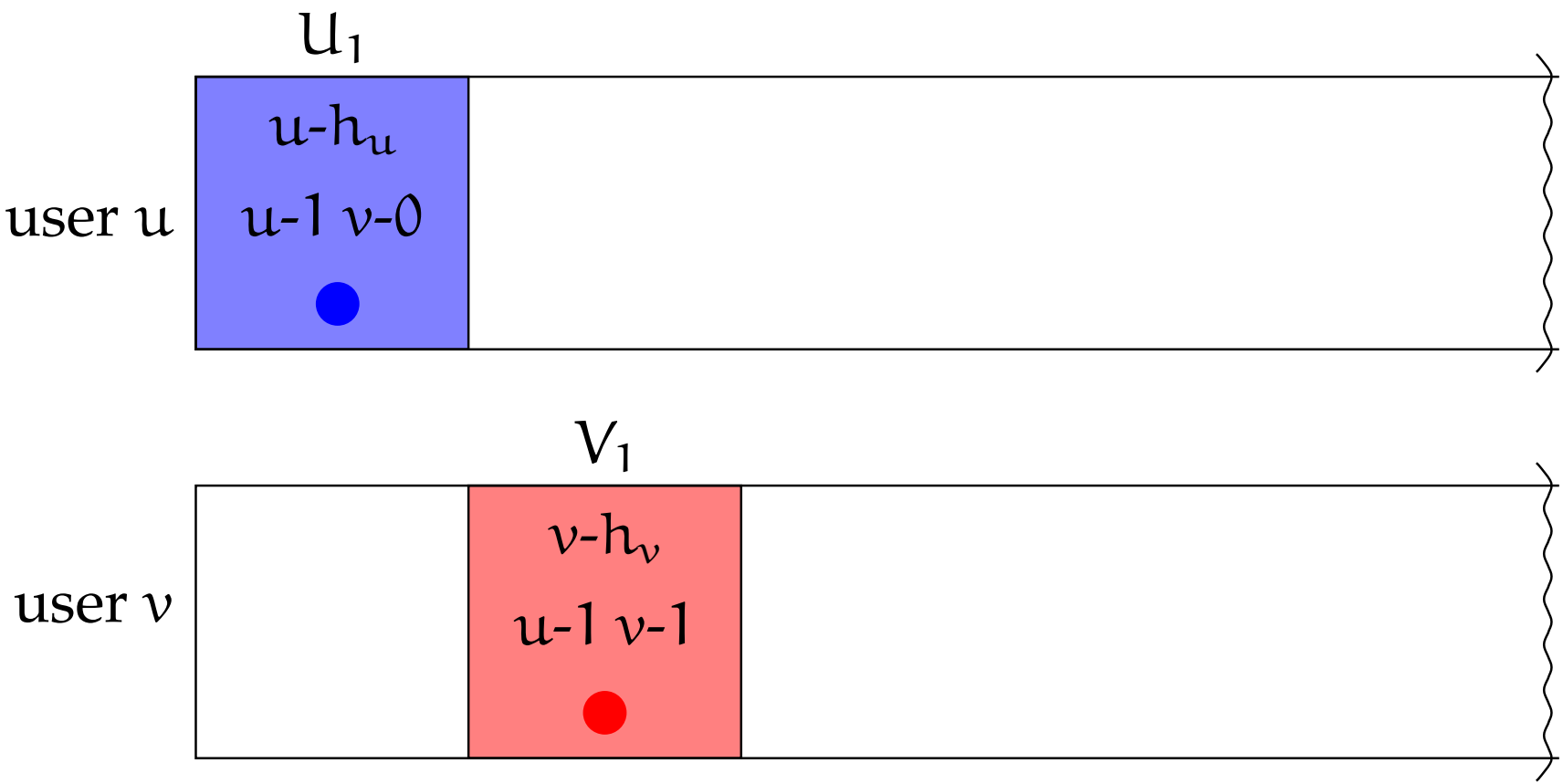
$$\textit{version structure:}\ \{\ \overbrace{u\text{-}h_u}^{\textbf{i-handle}}\ ,\ \overbrace{u\text{-}4\ v\text{-}2}^{\textbf{version vector}}\ \}_{K_u^{-1}}$$

- **Say $U \leq V$ iff no user has higher vers. # in $U$ than in $V$**
  - Idea: Unordered version structures signify an attack

# Solution 2: Serialized SUNDR

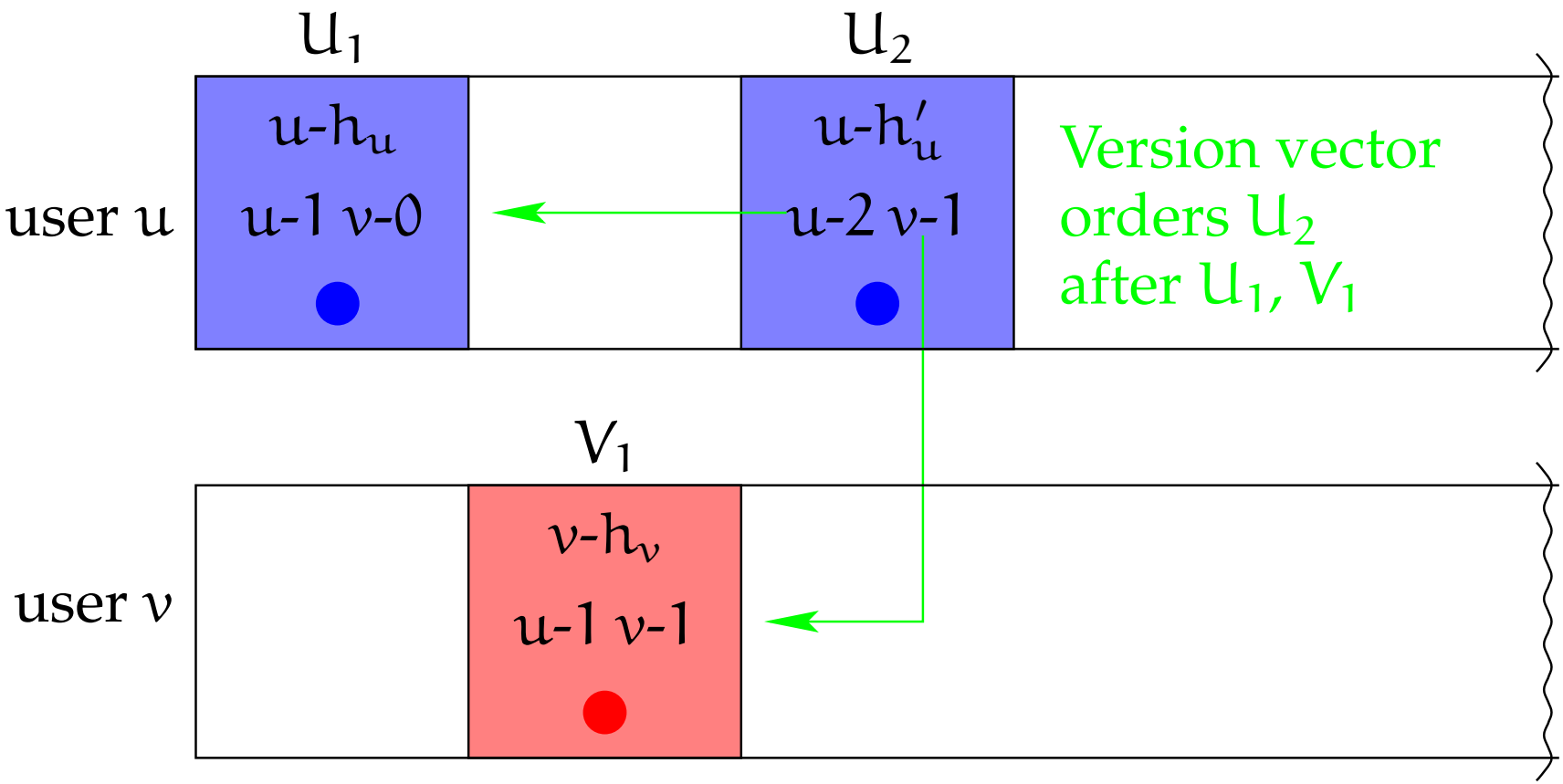
- **Still no concurrent updates**
- **Server maintains latest signed i-handle of each user/group in *version structure list* or VSL**
- **To fetch or modify a file, $u$'s client makes 2 RPCs:**
  - PREPARE: Locks FS, downloads and sanity-checks VSL
  - Client calculates & signs new version structure:
    $\{u\text{-}h_u, u\text{-}(n_u + 1)\ v\text{-}n_v\ \ldots\}_{K_u^{-1}}$
  - If modifying group i-handle, bump group version number:
    $\{u\text{-}h_u\ g\text{-}h_g, u\text{-}(n_u + 1)\ v\text{-}n_v\ \ldots\ g\text{-}(n_g + 1)\ \ldots\}_{K_u^{-1}}$
  - COMMIT: Uploads version struct for new VSL, releases lock

# Example: Honest server



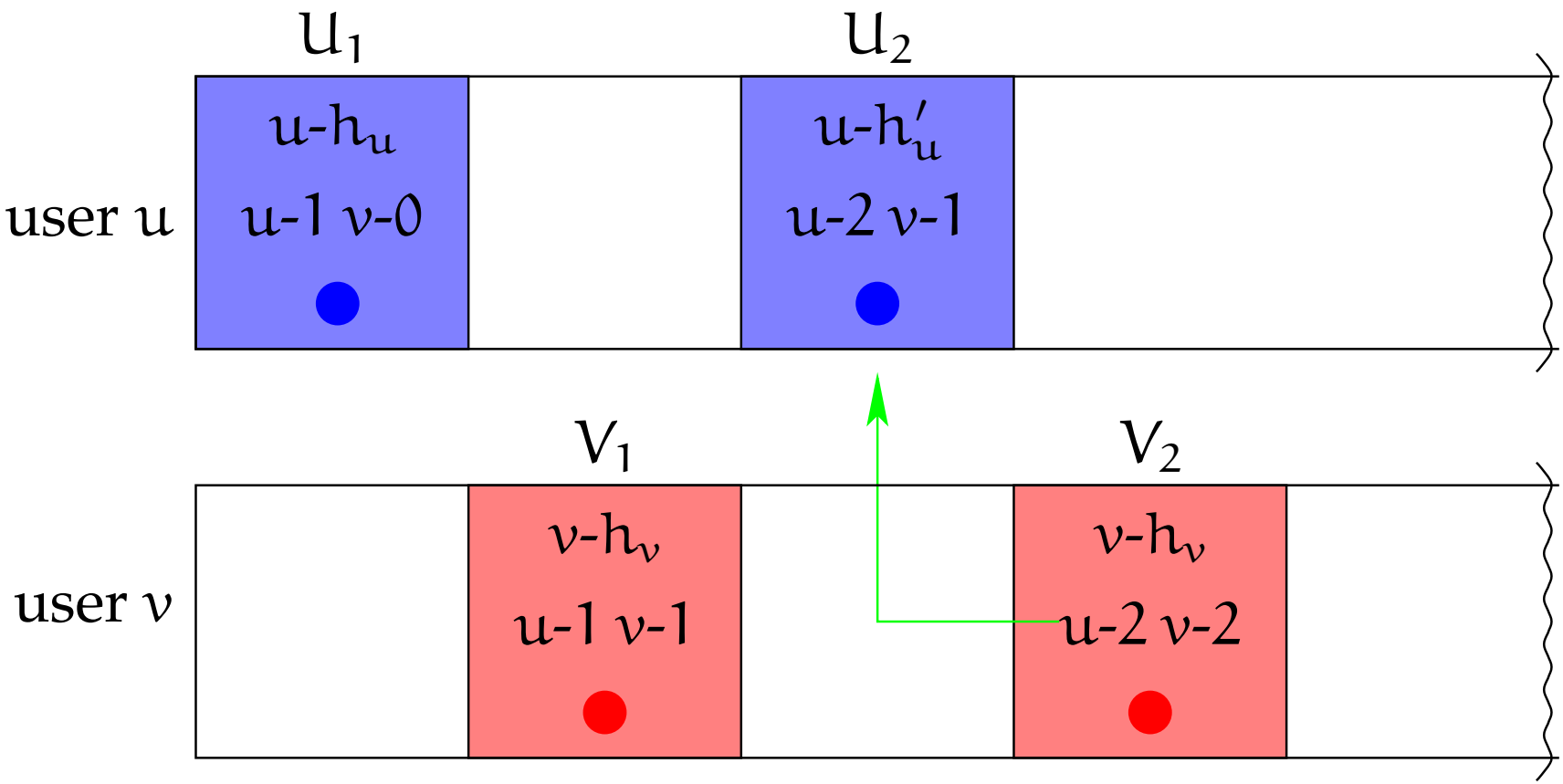

- **Users $u$ and $v$ each start at version 1 (sign $U_1$ & $V_1$)**

# Example: Honest server



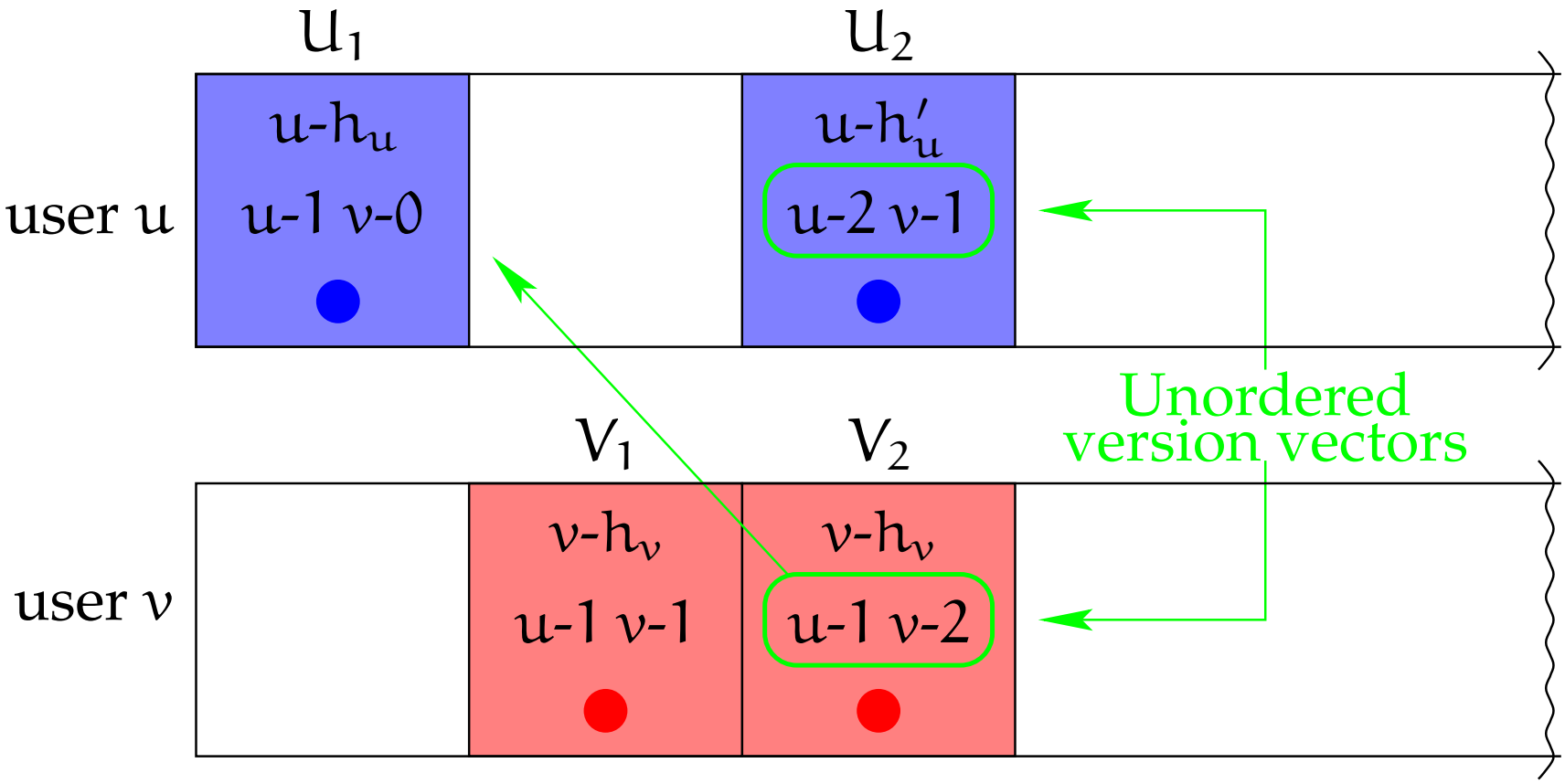

- **Users $u$ and $v$ each start at version 1 (sign $U_1$ & $V_1$)**
- **$u$ modifies file $f$, signs $U_2$ w. new i-handle $h'_u$**

# Example: Honest server



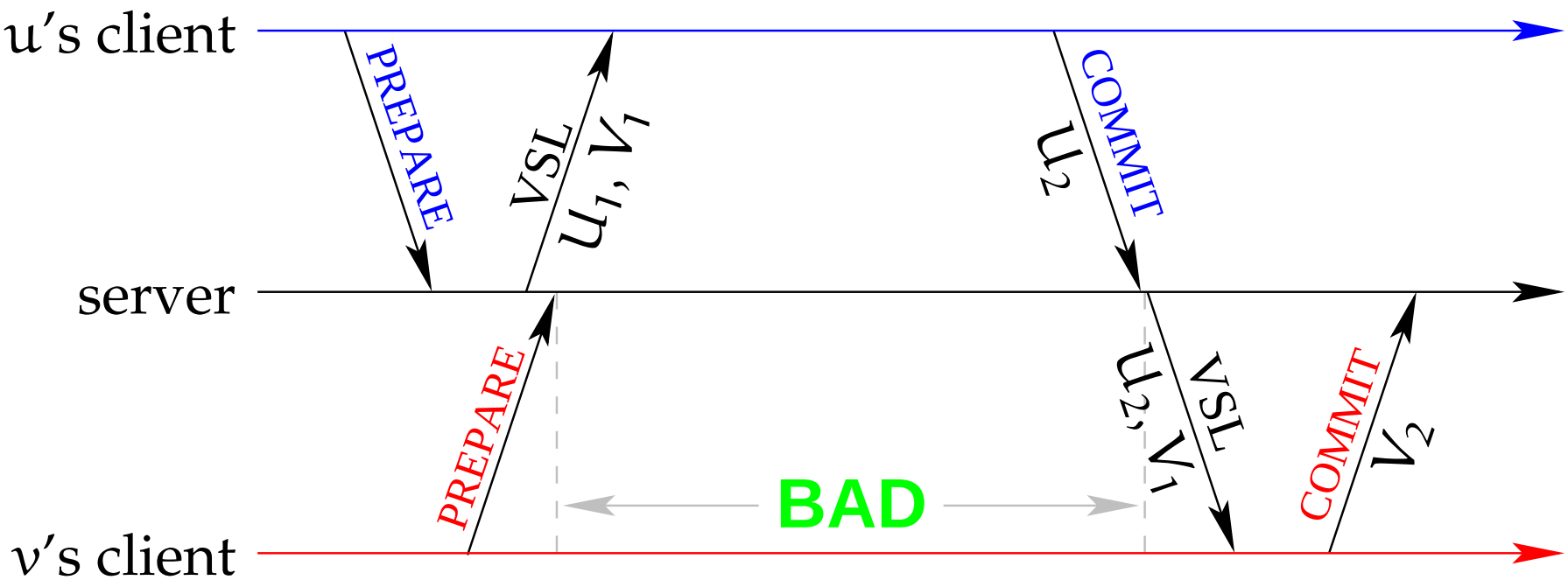

- **Users $u$ and $v$ each start at version 1 (sign $U_1$ & $V_1$)**
- **$u$ modifies file $f$, signs $U_2$ w. new i-handle $h'_u$**
- **$v$ fetches $f$, signs $V_2$ which reflects having seen $U_2$**

# Example: Malicious server



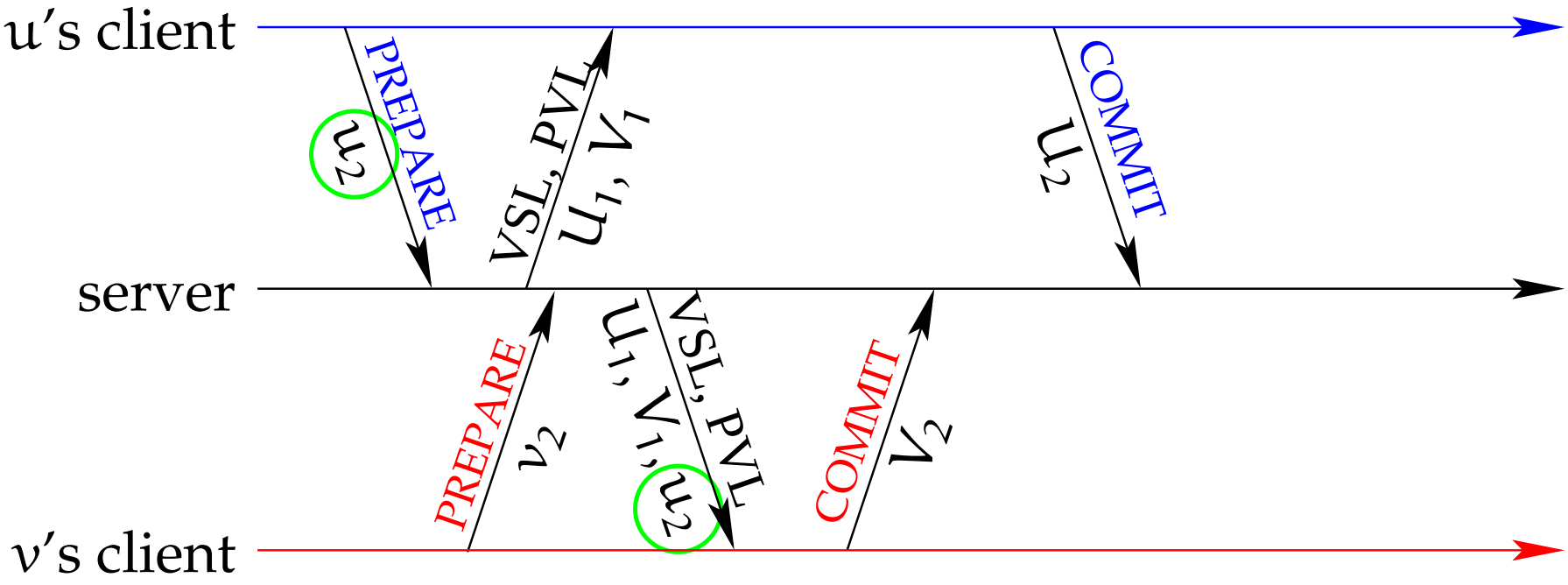

- **Suppose server hadn't shown $u$'s modification of $f$ to $v$**
- **Now $U_2 \not\leq V_2$ and $V_2 \not\leq U_2$**
  - $u$ or $v$ will detect attack upon seeing any future op by other

# Limitations of serialized SUNDR



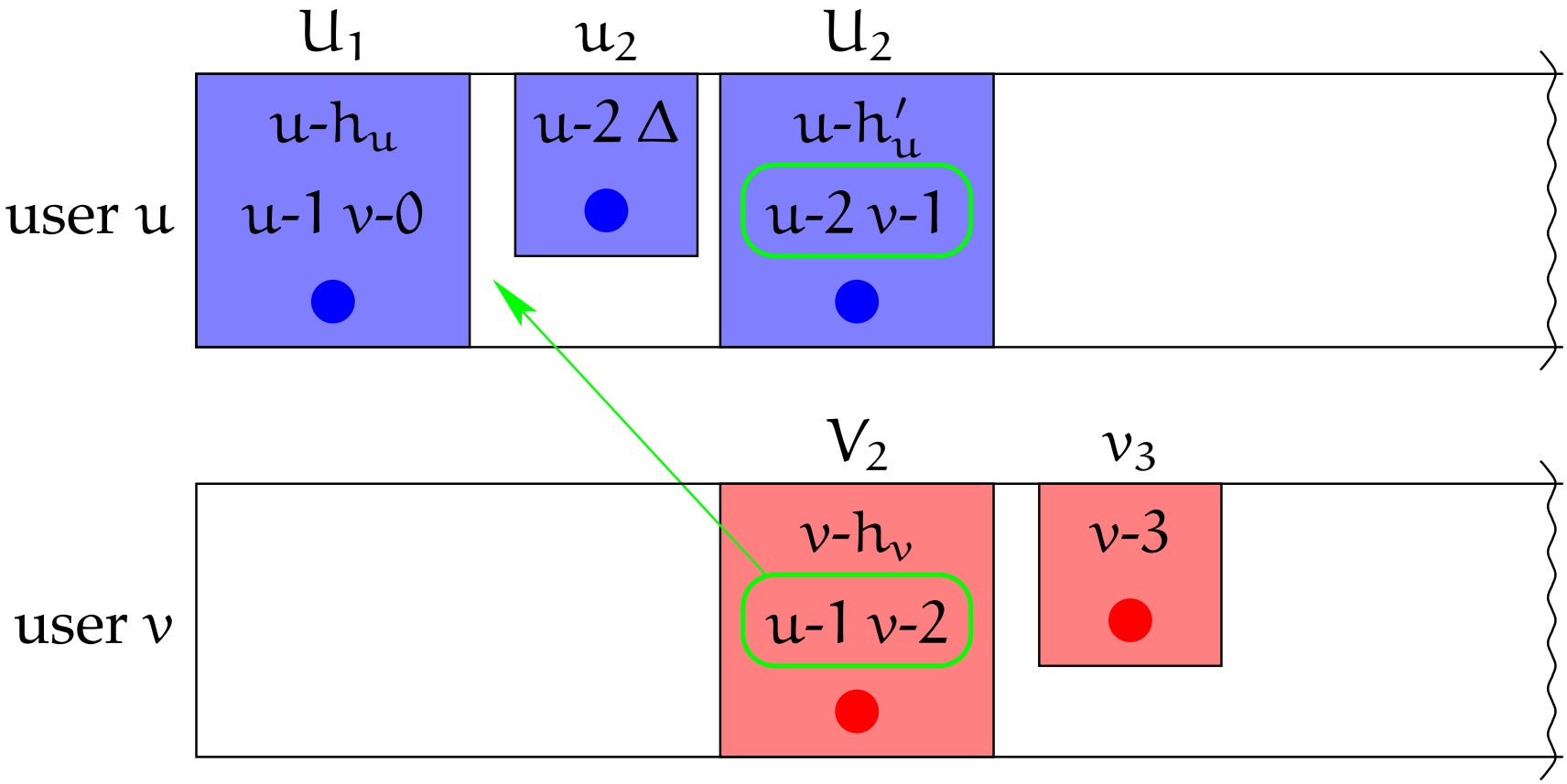

- **Honest server can only allow one operation at a time**
  - E.g., server must send $U_2$ to $v$ to prevent fork on last slide
  - Must wait *even if* $V_2$ *doesn't observe any changes made in* $U_2$
- **Without concurrency, get terrible I/O throughput**

# Solution 3: Concurrent SUNDR



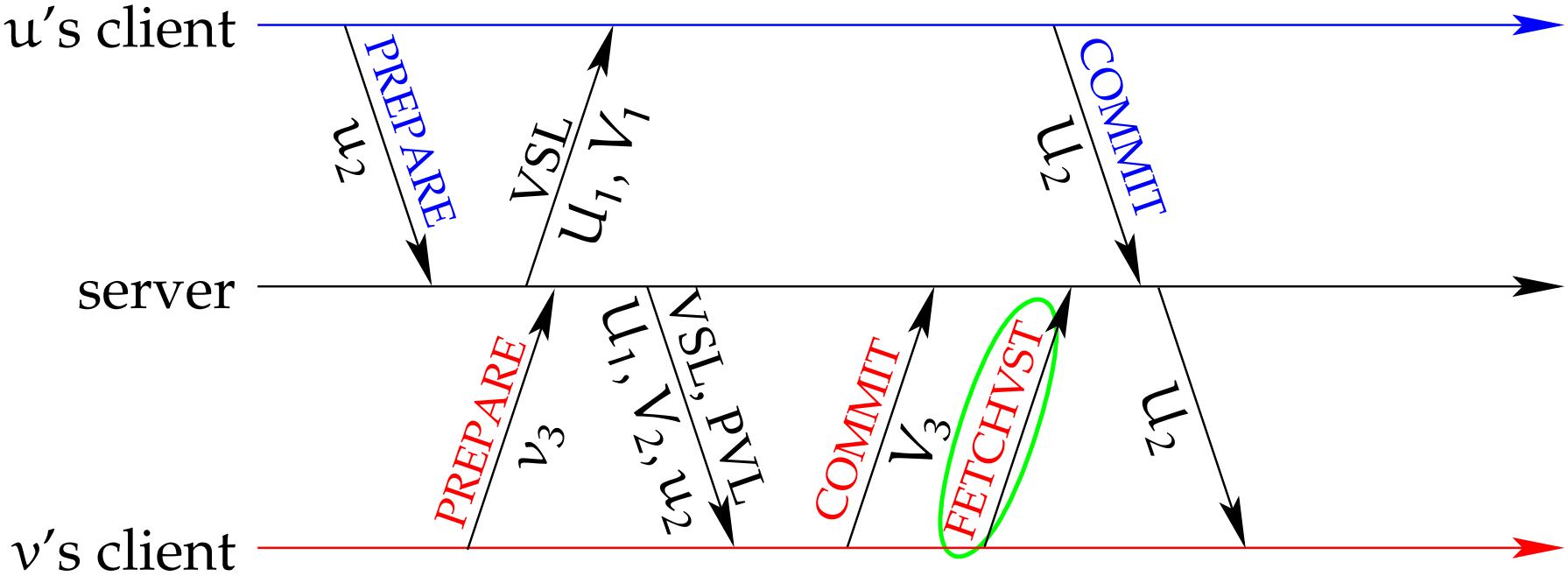

- **Pre-declare operations in signed *update certificates***
  - $u_2$ = "In version struct $U_2$, I intend to change file $f$ to i-hash $h$."
- **Server keeps uncommitted update certificates in *Pending Version List* or PVL, returns with VSL**
- **Idea: $v$ can compute $V_2$ w/o seeing $U_2$ if it sees $u_2$**

# Danger: Erasing evidence of fork attacks



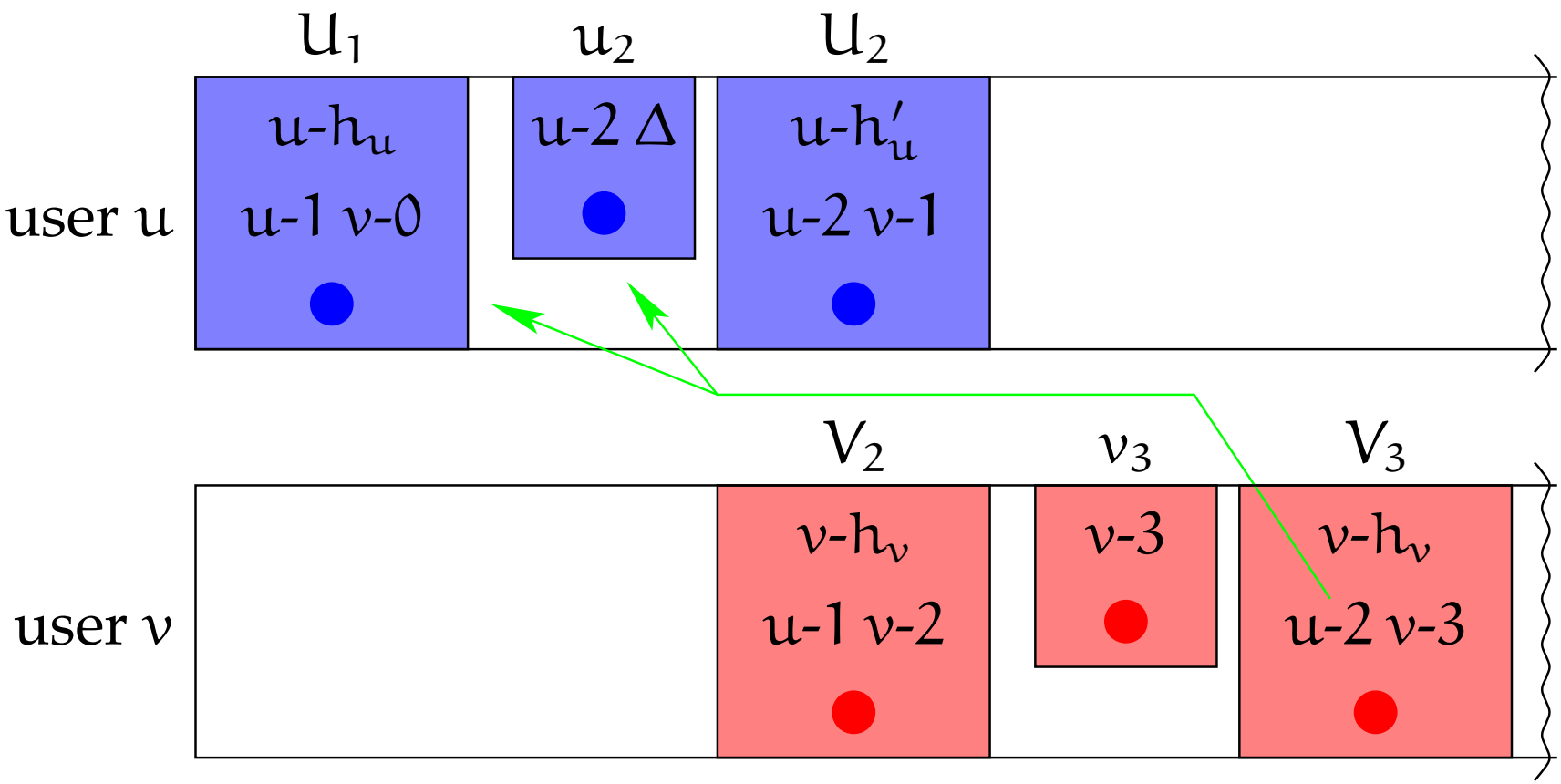

- **Let's revisit attack where $v$ missed modify of $f$ in $V_2$**
- **Say $v$ then PREPARES $v_3$ & server returns $U_1, V_2, u_2$**
  - Case 1: $v_3$ is fetching a file modified in $u_2$ (read-after-write)
  - Case 2: $v_3$ is not observing any changes declared in $u_2$

# Case 1: Read-after write conflict



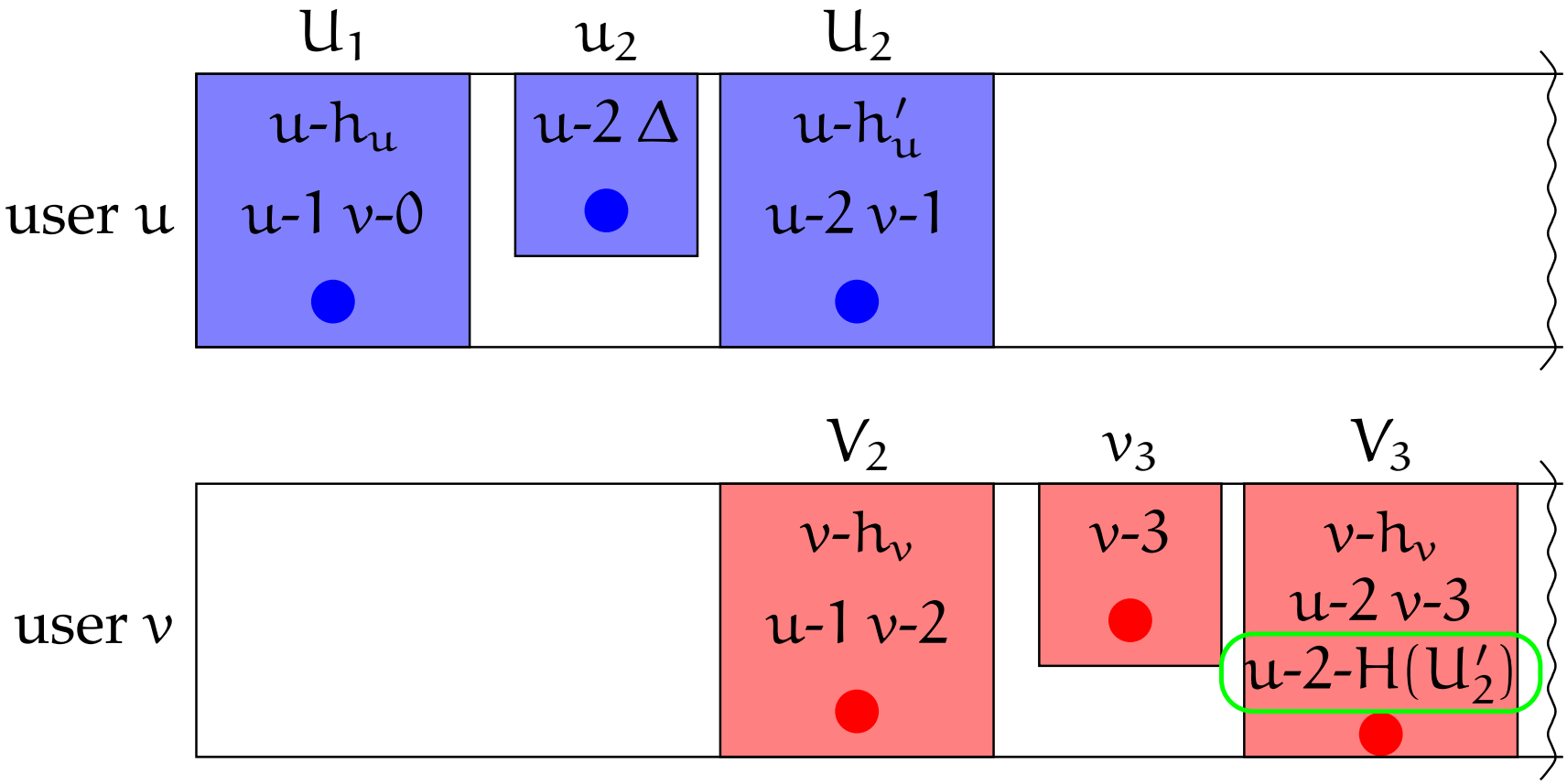

- **Must *not* show effects of $u_2$ to $v$'s application**
  - Recall: when $v$ sees change by $u$, should guarantee no attack
- **Wait for conflicting vstruct w. new FETCHVST RPC:**
  - Only required when fetching concurrently modified file
  - Example: $v$ will detect attack given $U_2$ because $V_2$ still in VSL, yet $U_2 \not\leq V_2$ and $V_2 \not\leq U_2$

# Case 2: No read-after-write conflict




- **Don't want to issue/wait for FETCHVST if no conflict**
- **Problem: $v$ will sign $V_3$ such that $U_2 \leq V_3$**
  - VSL is once again ordered, evidence of attack erased

# Solution: Reflect pending updates in vstructs

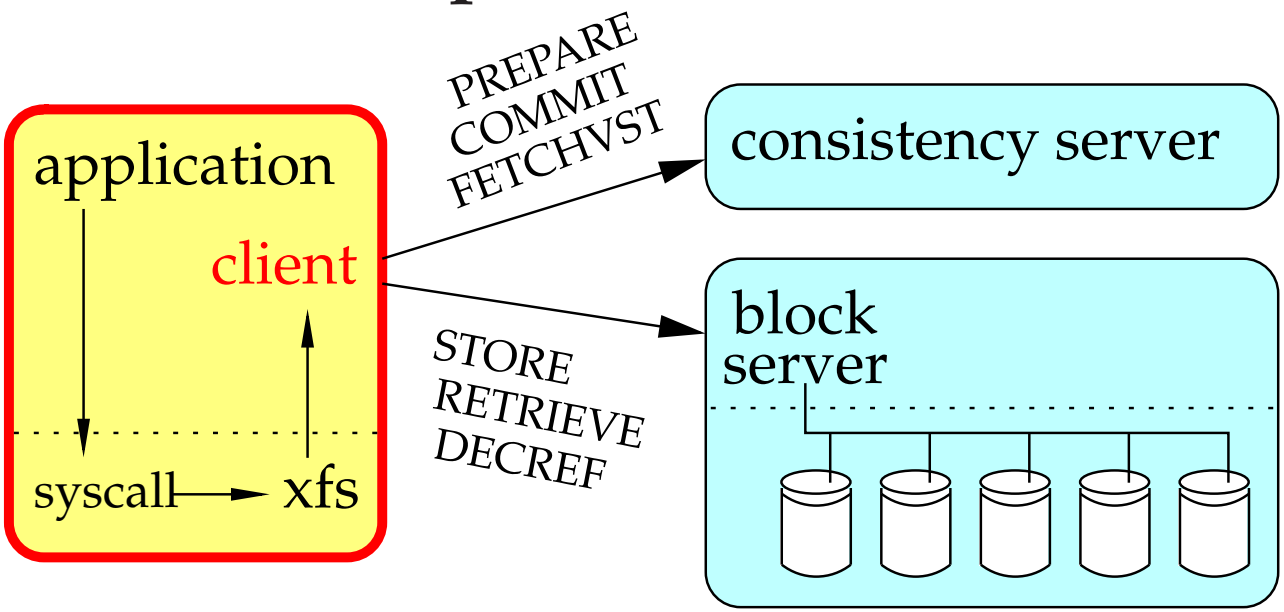



- **Vstruct includes hashes of other anticipated vstructs**
  - Omit i-handles so contents deterministic given order of PVL
- **Redefine $\leq$ to require that hashes match**
  - E.g., $U_2 \not\leq V_3$, because $V_3$ contains hash of $U'_2 = \{u\text{-}2\ v\text{-}2\} \neq U_2$

# Recovery

- **How to recover if server destroys data?**
  - E.g., smashes or wipes hard disk
- **People already expect disks to die & back up**
- **With SUNDR, no need to trust the backup!**
  - Could dump clients' cache contents to new server!
  - Signed version vectors ordered… use most recent available one for each user/group (will be widely cached)
  - Everything else indexed by hash… simply load up new server with data in cache—even files you could only read

# Implementation

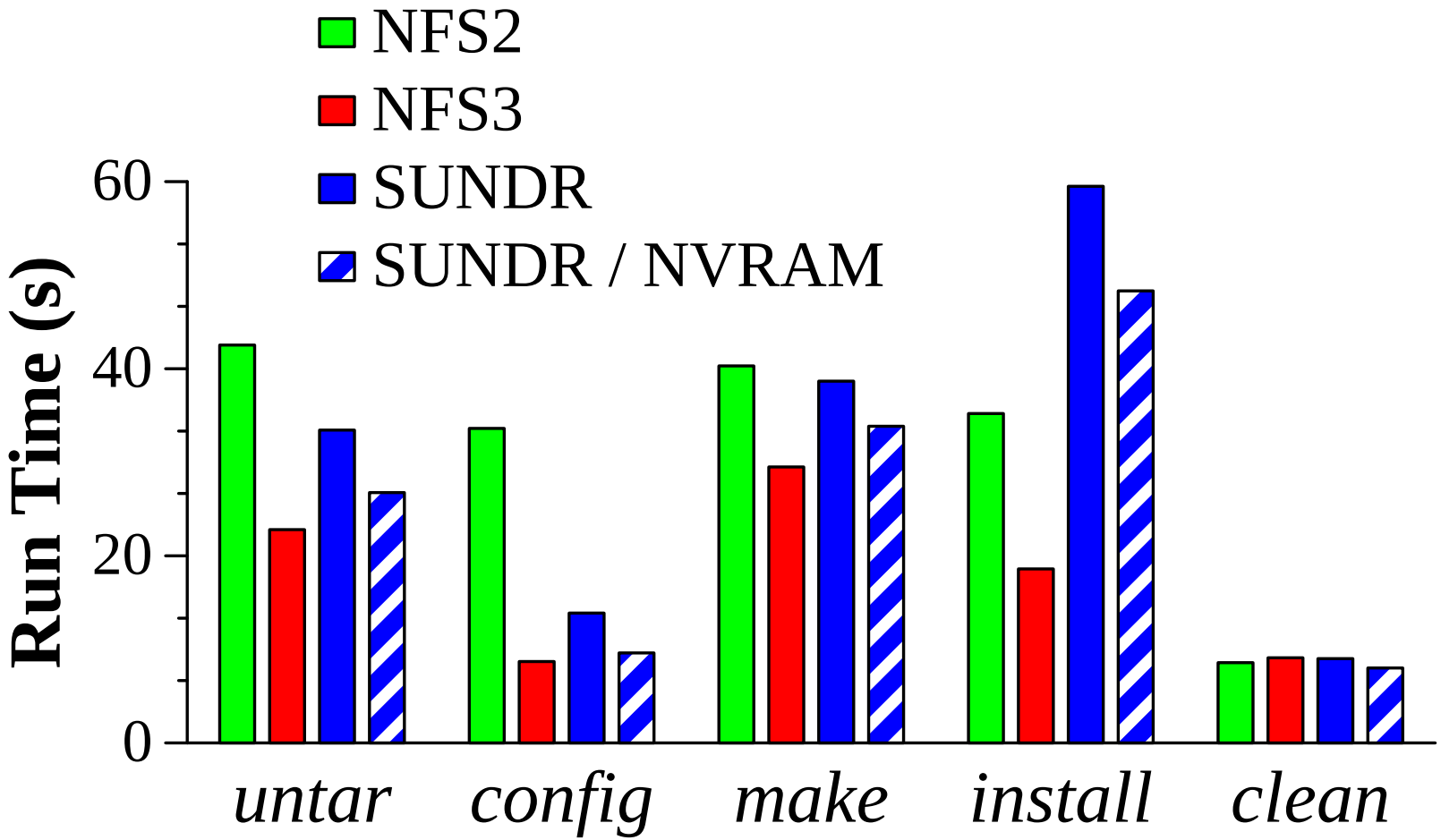



- **Client based on xfs device driver**
  - xfs part of Arla, a free AFS implementation
  - Designed for AFS-like semantics
- **Server split into two daemons**
  - *Consistency server* handles update certs, version structs
  - *Block server* stores bulk of data
  - Can run on same or different machines

# Further optimizations

- **i-handles really hash plus some deltas**
  - Amortizes recomputing hash tree over multiple ops
- **Include multiple fetches/modifies in one operation**
- **i-tables are Merkle B+-trees**
- **Group i-tables add yet another level of indirection**
  - No need to change group i-table if same user writes group-writable file twice
- **Concurrent modifications of same group i-table**
  - Possibly many files in a group—shouldn't serialize access
  - Users fold each other's forthcoming changes into i-table
  - Safety comes from careful definition of "$\leq$"

# SUNDR: Security *and* usable performance




- **Benchmark: unpack, build, install emacs 20.7**
  - 3 GHz Pentium IVs connected by 100 Mbit/sec Ethernet
  - Index on 4 15K RPM SCSI disks, logs on 7,200 RPM IDE disks