How to Protect your Data
by Eliminating Trusted Storage Infrastructure

David Mazières

Stanford University

work performed in collaboration with

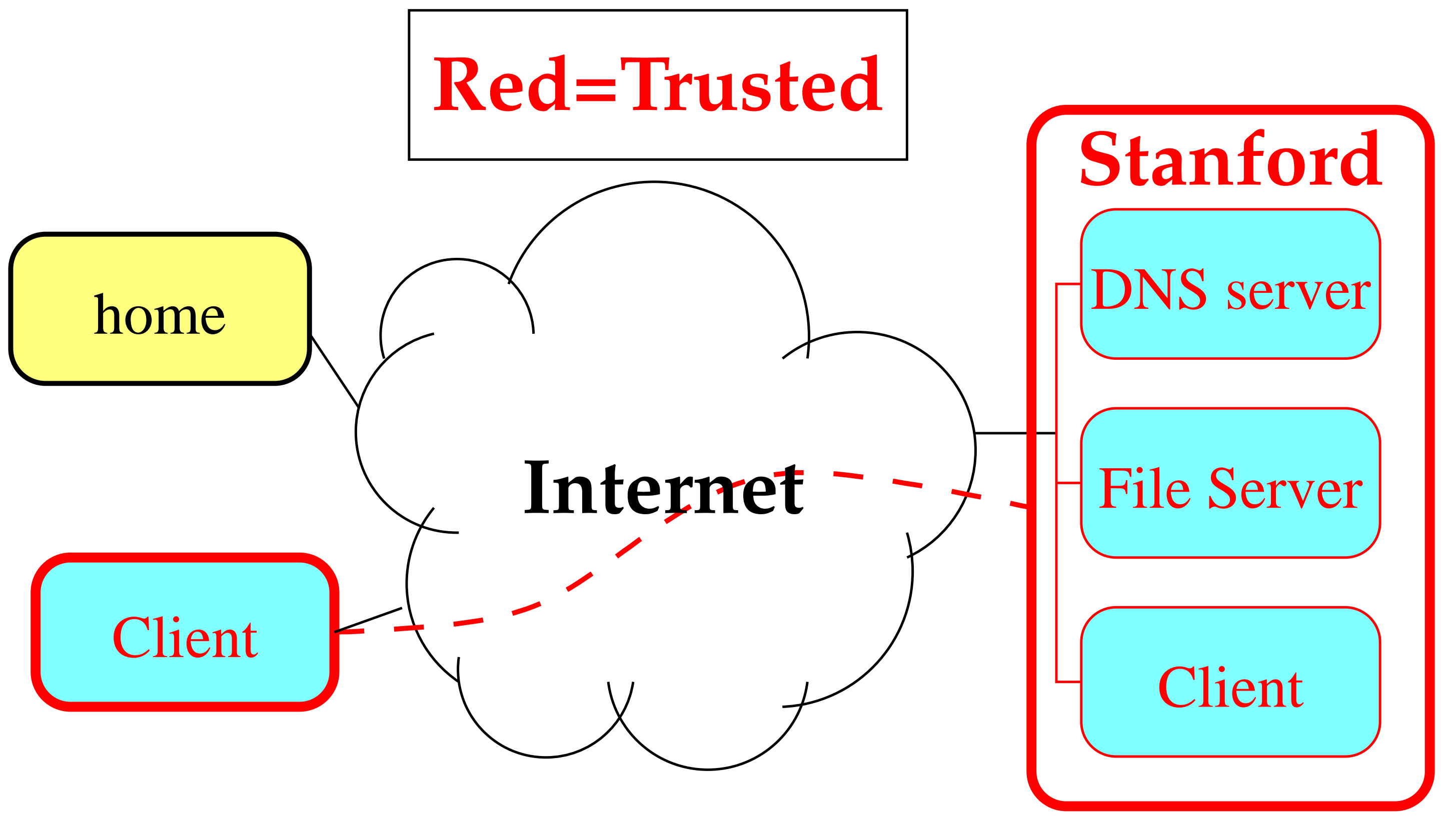
Jinyuan Li, Maxwell Krohn, Dennis Shasha,

Siddhartha Annapureddy, Benjie Chen, Frank Dabek, Yevgeniy Dodis, Michael Freedman, Kevin Fu,

Daniel Giffin, Frans Kaashoek, Michael Kaminsky, Petar Maymounkov, Robert Morris,

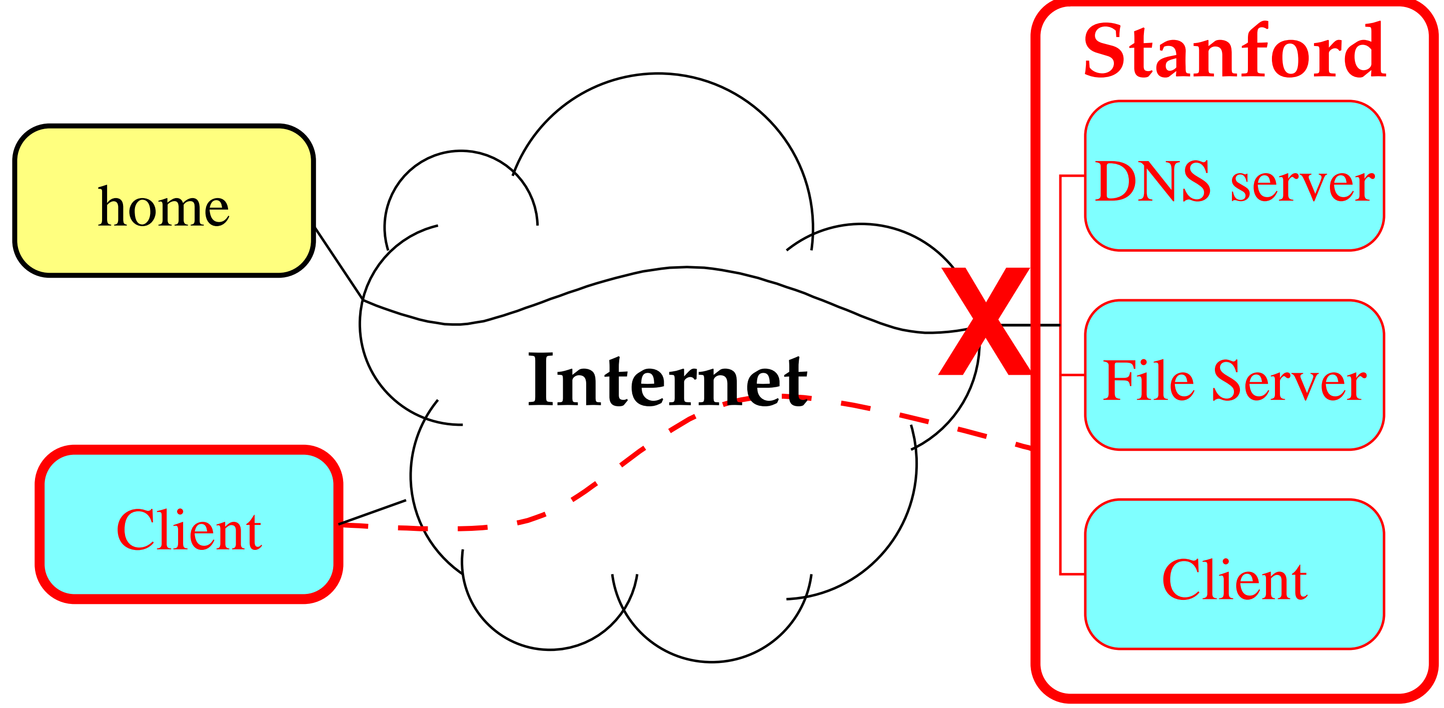
Athicha Muthitacheroen, Antonio Nicolosi, George Savvides, Emmett Witchel, Nickolai Zeldovich

Security today: The fence approach



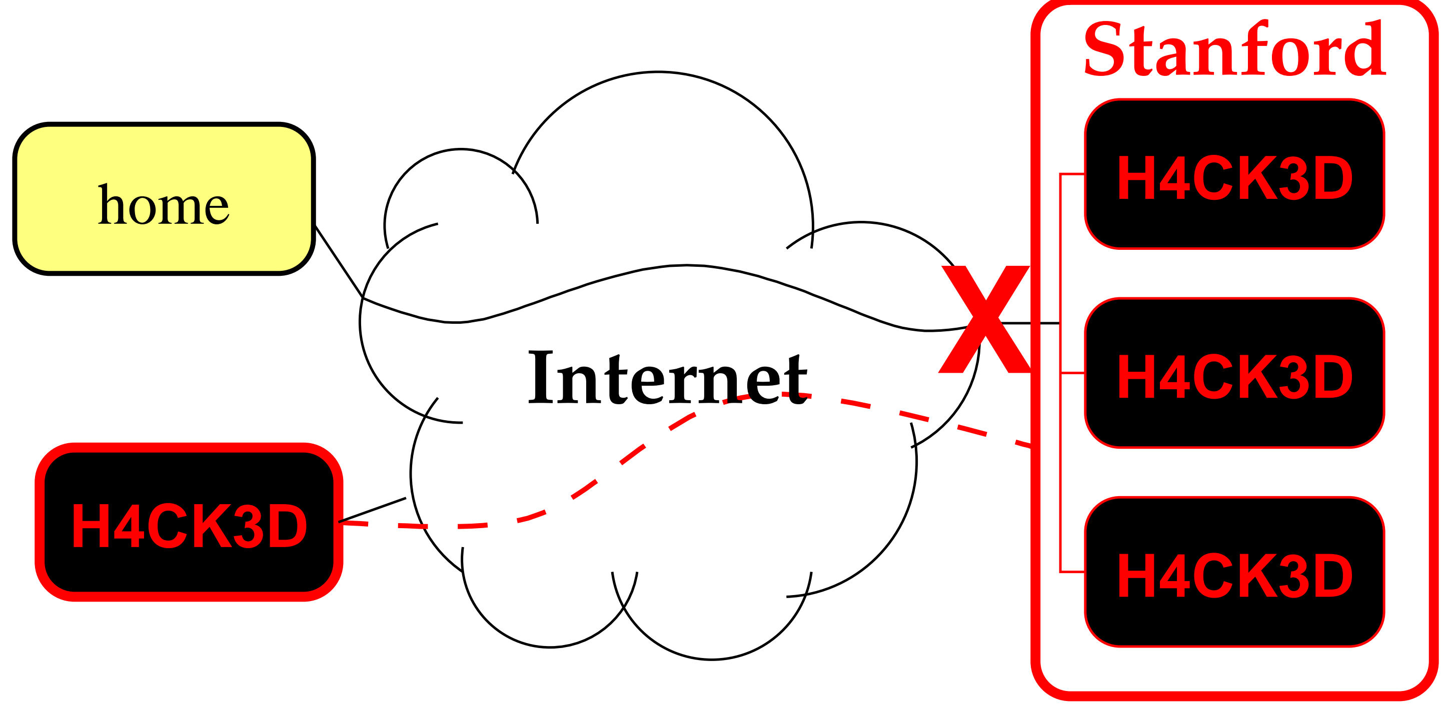
- **Seal off your server & clients with a firewall**
  - Virtualize to remote clients using VPNs
- **Simplifies administration (coarse-grained policy)**

Limitations of the fence approach



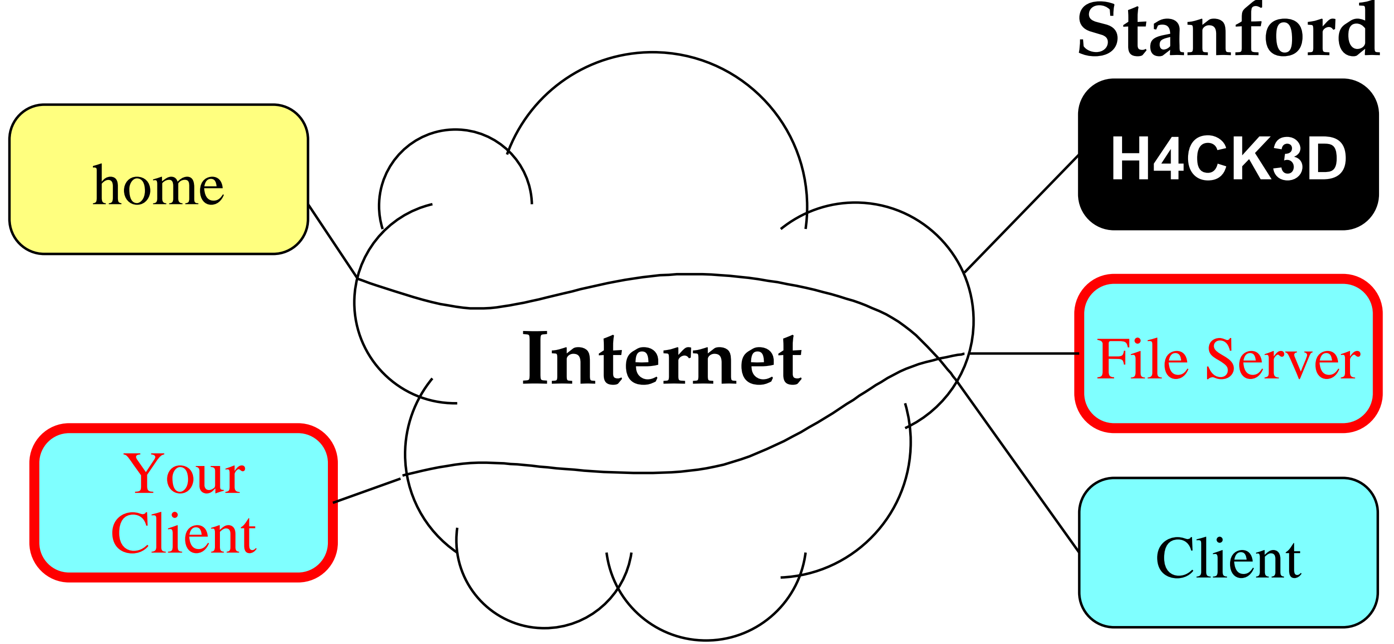
- **Problem: Big fences mean vague security policies**
  - Prohibit some legitimate behavior (a pain for users)
  - Permit some dangerous interactions (insufficiently secure)
- **Perimeter security is all-or-nothing**
  - Breaches or insider attacks can be catastrophic

Limitations of the fence approach



- **Problem: Big fences mean vague security policies**
  - Prohibit some legitimate behavior (a pain for users)
  - Permit some dangerous interactions (insufficiently secure)
- **Perimeter security is all-or-nothing**
  - Breaches or insider attacks can be catastrophic

Alternative: End-to-end security



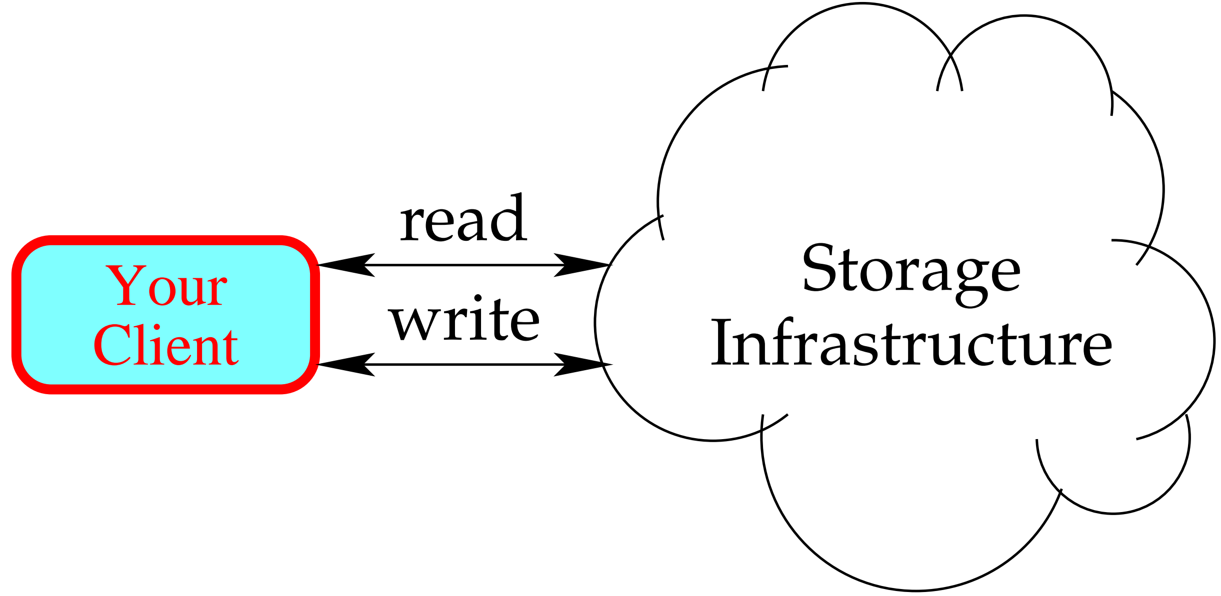
- **Shrink the diameter of fences to reduce trust**
  - Tightly enclose entities making security-relevant decisions
  - Fewer weak points (a.k.a. small TCB—longstanding goal)
- **Lift unnecessary restrictions on users**
  - Accommodate functionality that doesn't fit the fence model

Challenges in achieving end-to-end security

1. **Re-factor applications, pushing trust to end points**
   - Often requires user-visible changes (e.g., to capture intent)
   - Example: No secure drop-in replacement for NFS
2. **Devise novel crypto algorithms or protocols**
3. **Engineer practical systems (e.g., release software)**
   - Test the usability of an idea
   - Make a qualitative impact on people's computing
4. **Harden the endpoints**

Protecting data

- **This talk: Apply approach to protecting data in files**
- **Help applications that rely on files (most)**
- **Capitalize on narrow interface of file systems:**



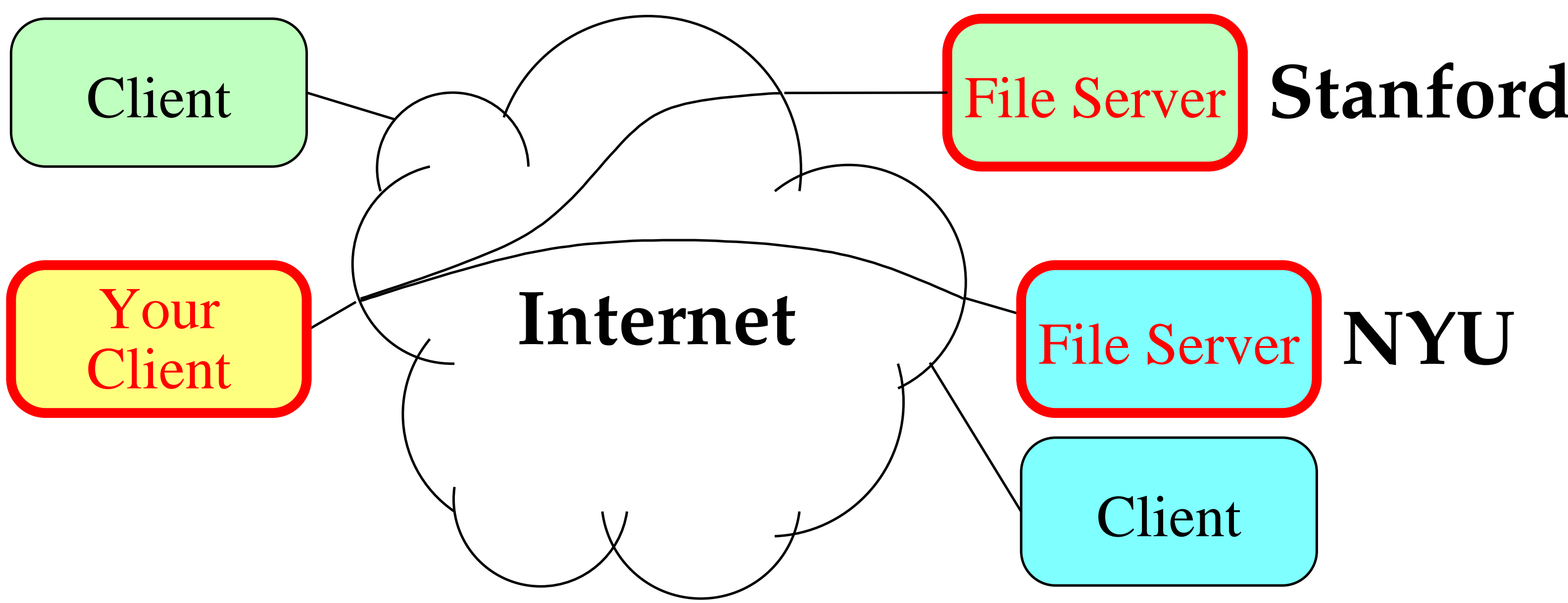
  - Can specify precise end-to-end security properties
  - Can even prove theorems about file system protocols

Outline

- **SFS: Trust only the endpoints – your client & server**
  - Re-factor security to exclude key management [SOSP'99]
  - Novel protocols for authentication [NDSS'03,SOSP'03]
  - Practical software [USENIX'01,SOSP'01,USENIX'03]
- **SFSRO: Eliminate trust in server [OSDI'00/TOCS'02]**
  - Solves secure content distribution – not general-purpose FS
- **SUNDR: True end-to-end file security (bulk of talk)**
  - Clients check for themselves no unauthorized modifications
  - Can detect problems even if attacker completely controls server! (SUNDR is first file system to achieve this property)
  - Even if server colludes with bad users
  - Novel protocol [PODC'02] & system [OSDI'04]

SFS

SFS (Self-certifying File System)



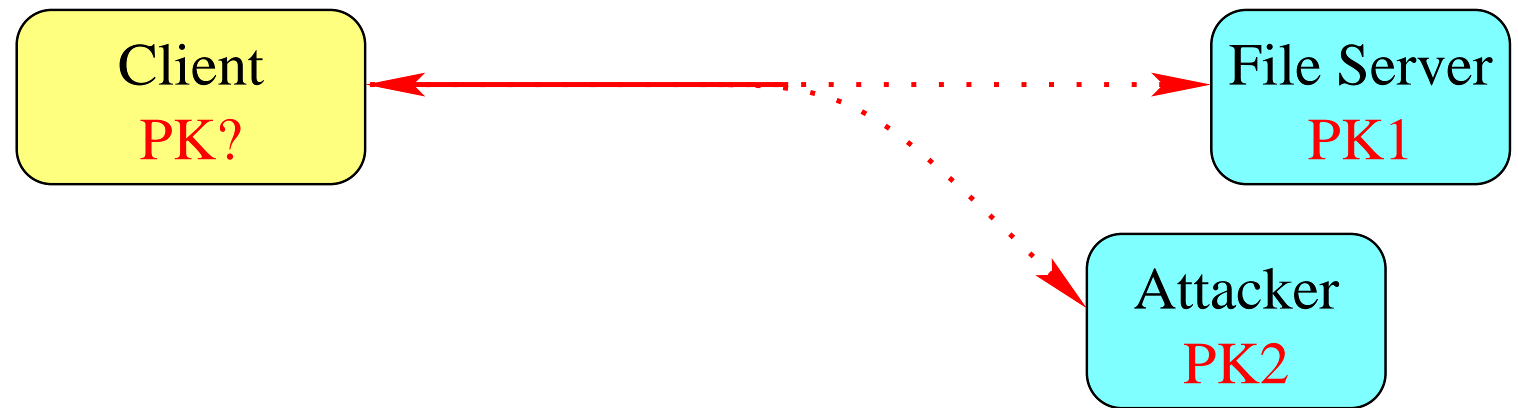
- **Shrink the fence down to the client and server**
  - No need to trust network, DNS, other clients, CAs, etc.
- **End-to-end security enables new functionality**
  - Makes administrative boundaries irrelevant (e.g., simultaneous access to NYU and Stanford from anywhere)

"Just adding" security is hard

- **Previous file systems didn't capture users' intents**
- **User interface looks like:** `/net/scs.stanford.edu/dm`
- **Say my intent is to talk to server in my office**
- **In big fence world:**
  - Trust Verisign to identify Stanford
  - Trust Stanford to assign this name to my server
- **How to move Verisign & Stanford outside the fence?**
  - Can't with this interface
  - Really want `/net/machine-in-my-office/dm`

Re-factoring security in SFS

- **Problem goes away if client knows server's public key**



- **Often can get keys w/o trusting Verisign or Stanford**
  - E.g., Use passwords to get public keys securely from servers
  - But how to express public key to file system client software?
- **Idea: Put the public key in the pathname**

  `/sfs/@sfs.stanford.edu,bzcc5hder7cuc86kf6qswyx6yuemnw69/dm/`

  - Symbolic links save users from seeing these names

SFSRO

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

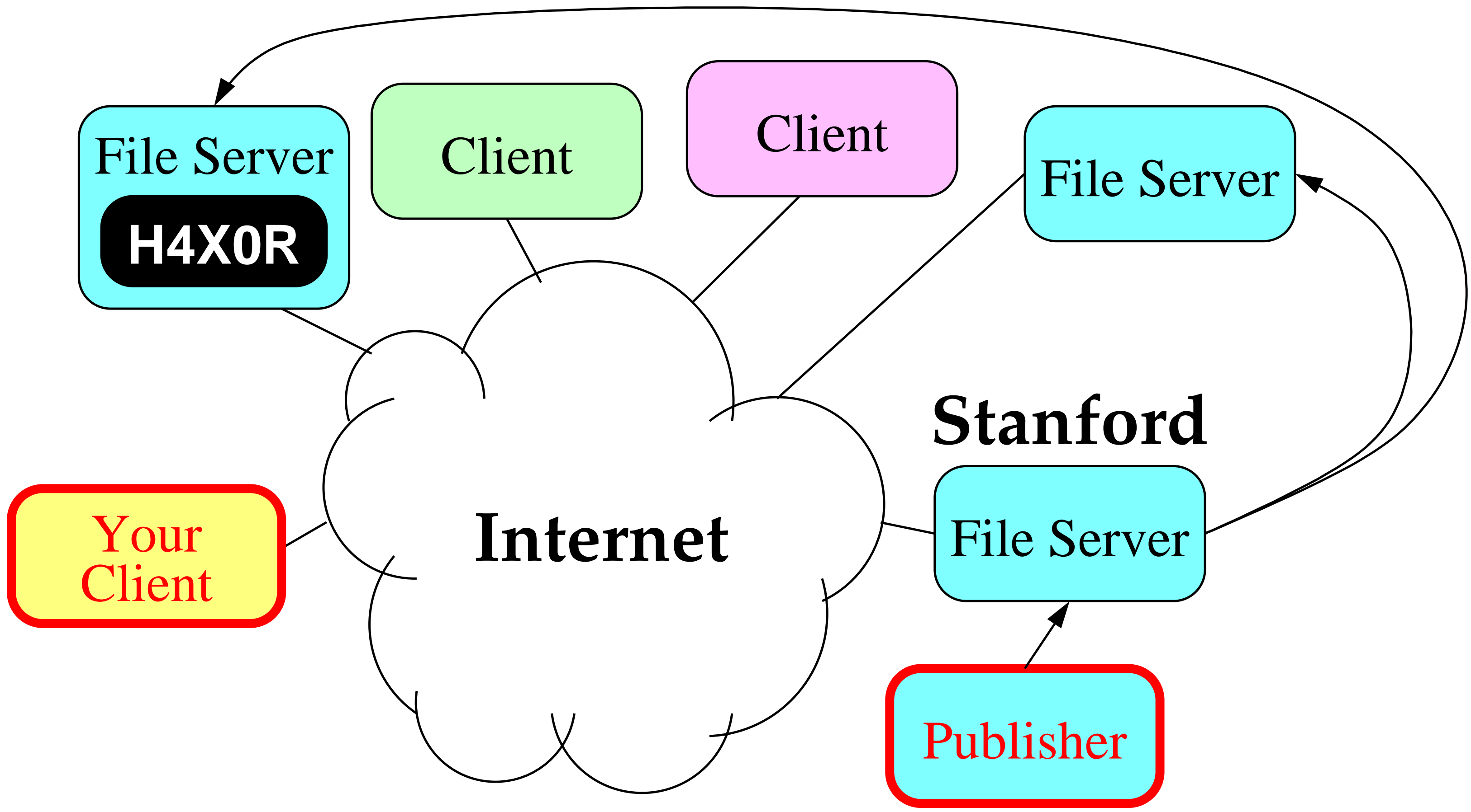
Signing individual files

- **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?**
  - What about directory name? Any context used to interpret file?
- **How do you know users will check signature?**

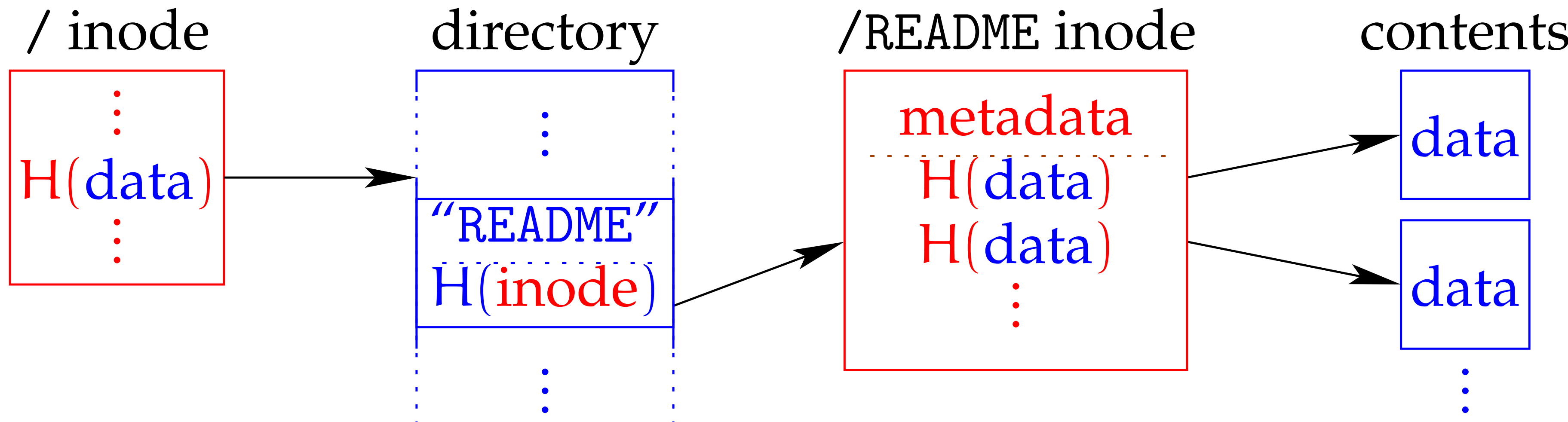
SFSRO solution: Signing whole file systems



- 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

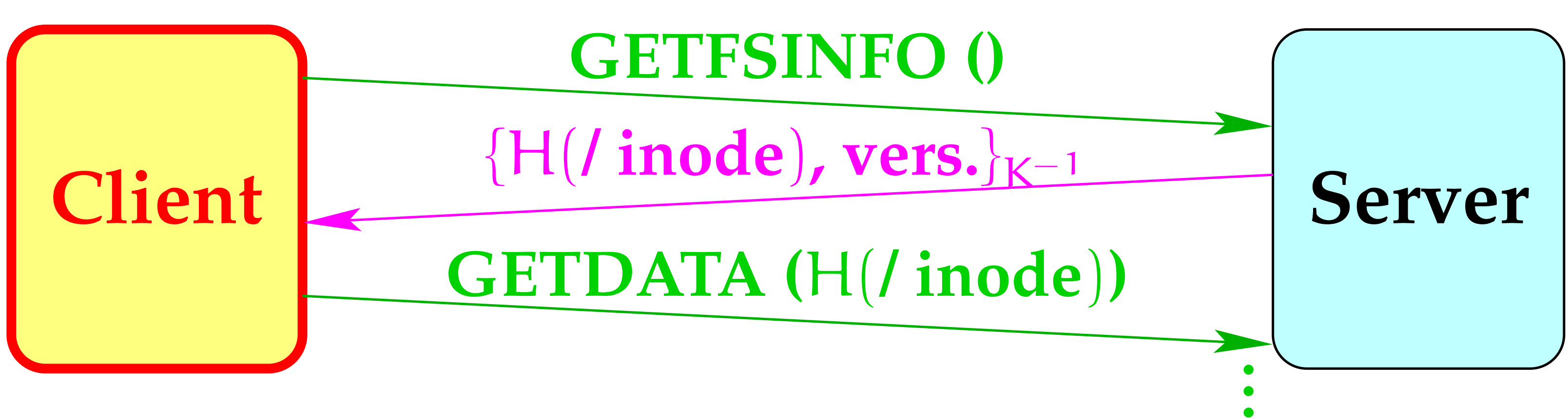
- **Can't just sign raw disk image (too big)**
  - Users may want to download and verify only a few files
- **Idea: Index all data & metadata by cryptographic hash**



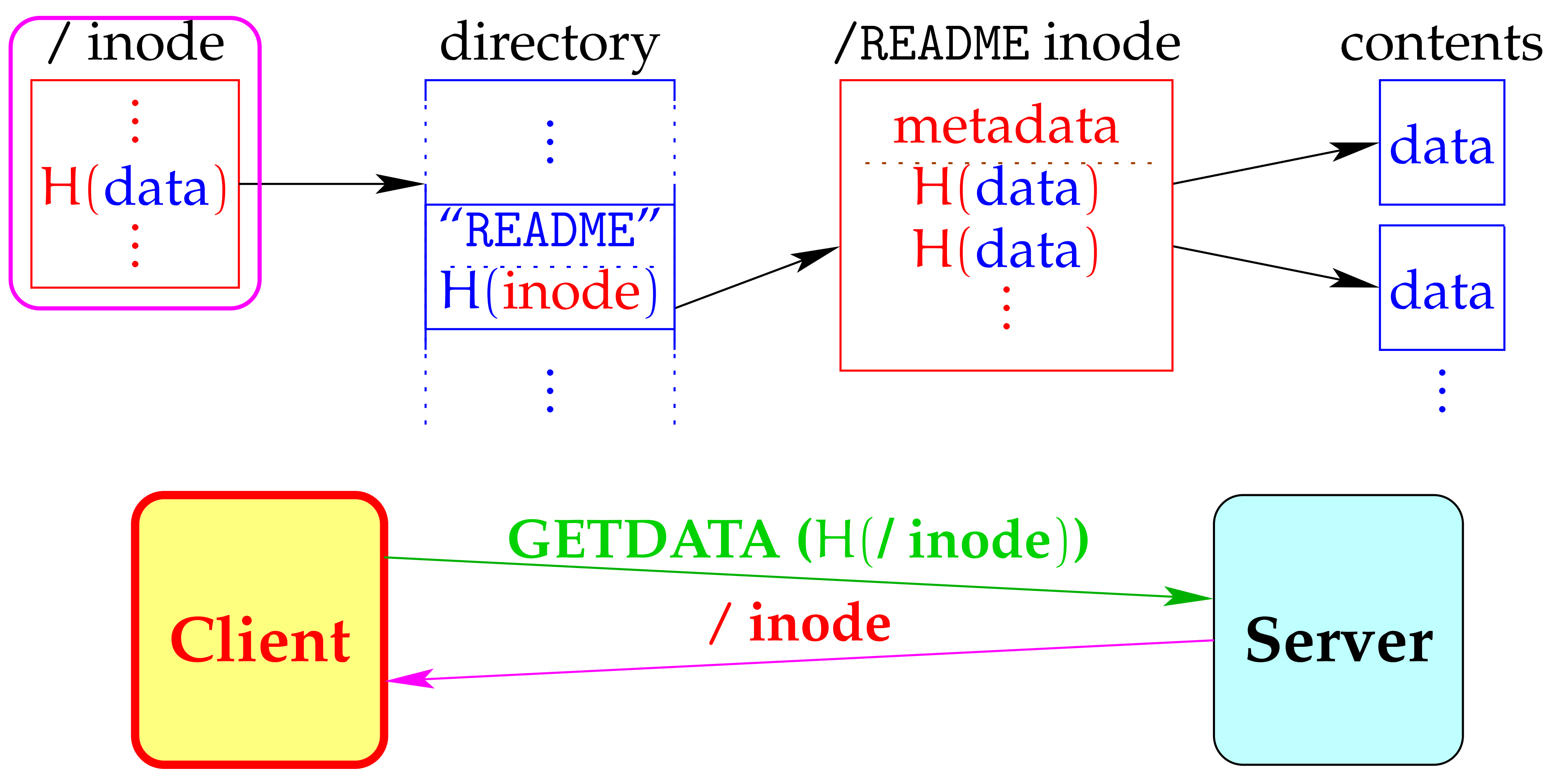
  - H is a collision-resistant hash function w. fixed-size output
- **Publisher signs hash of root inode**
- **Idea influenced many systems (CFS, Venti, . . .)**

SFSRO Protocol

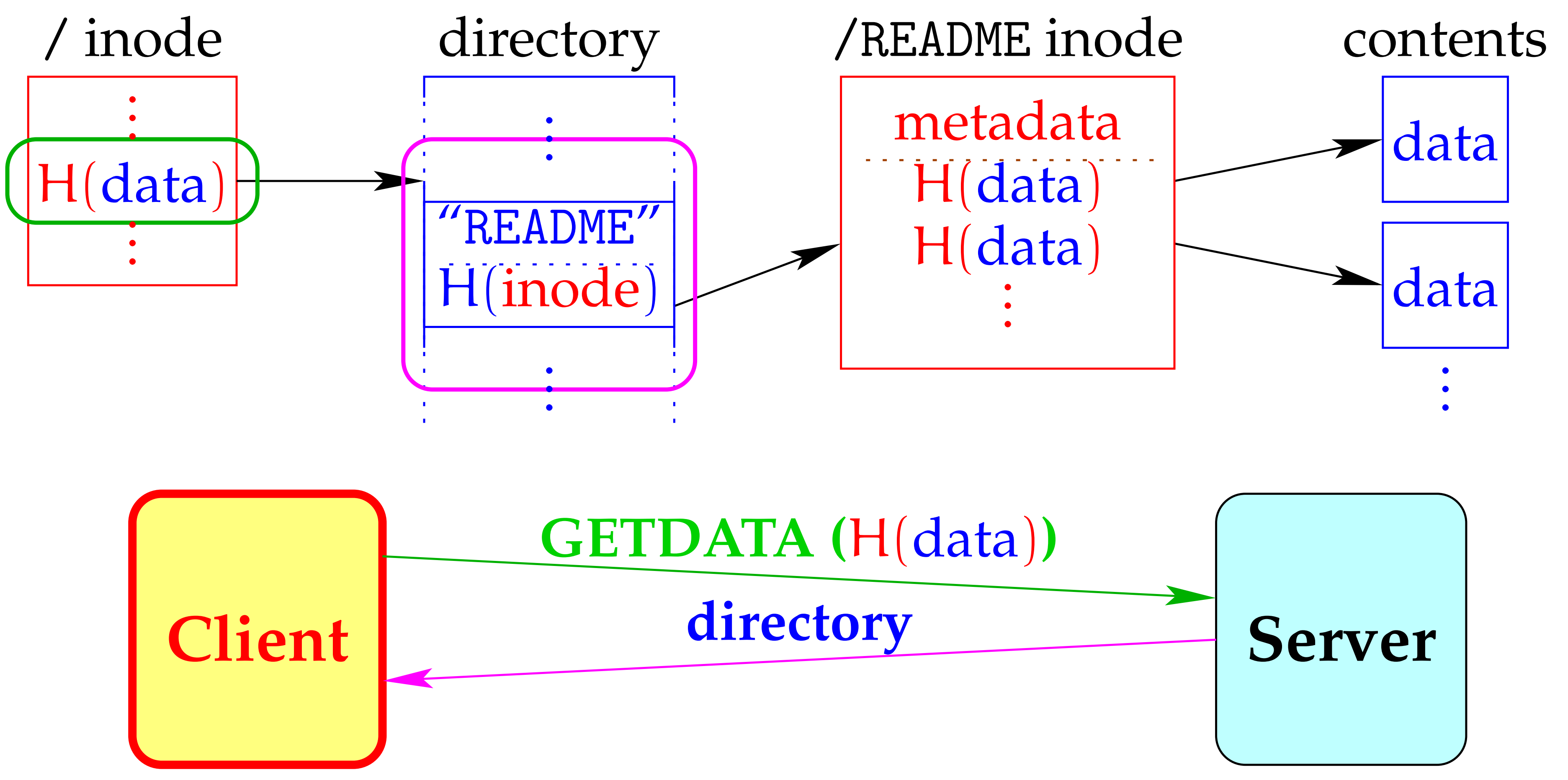
- **GETFSINFO () – Get signed hash of root directory**
- **GETDATA (*hash*) – Get block with *hash* value**
- **Example: To read file `/README`**
  - First get signed hash, then walk down tree



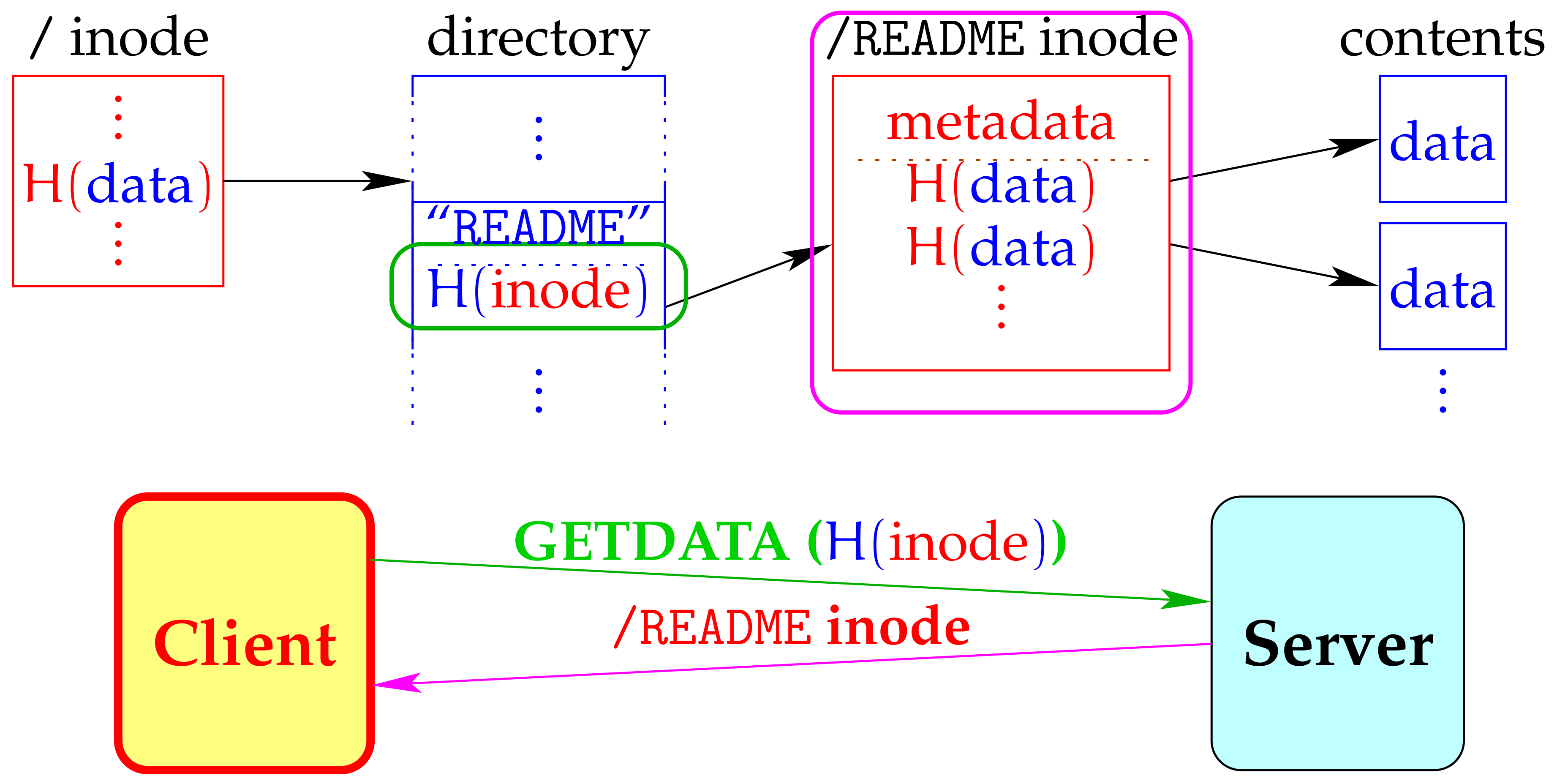
SFSRO Protocol



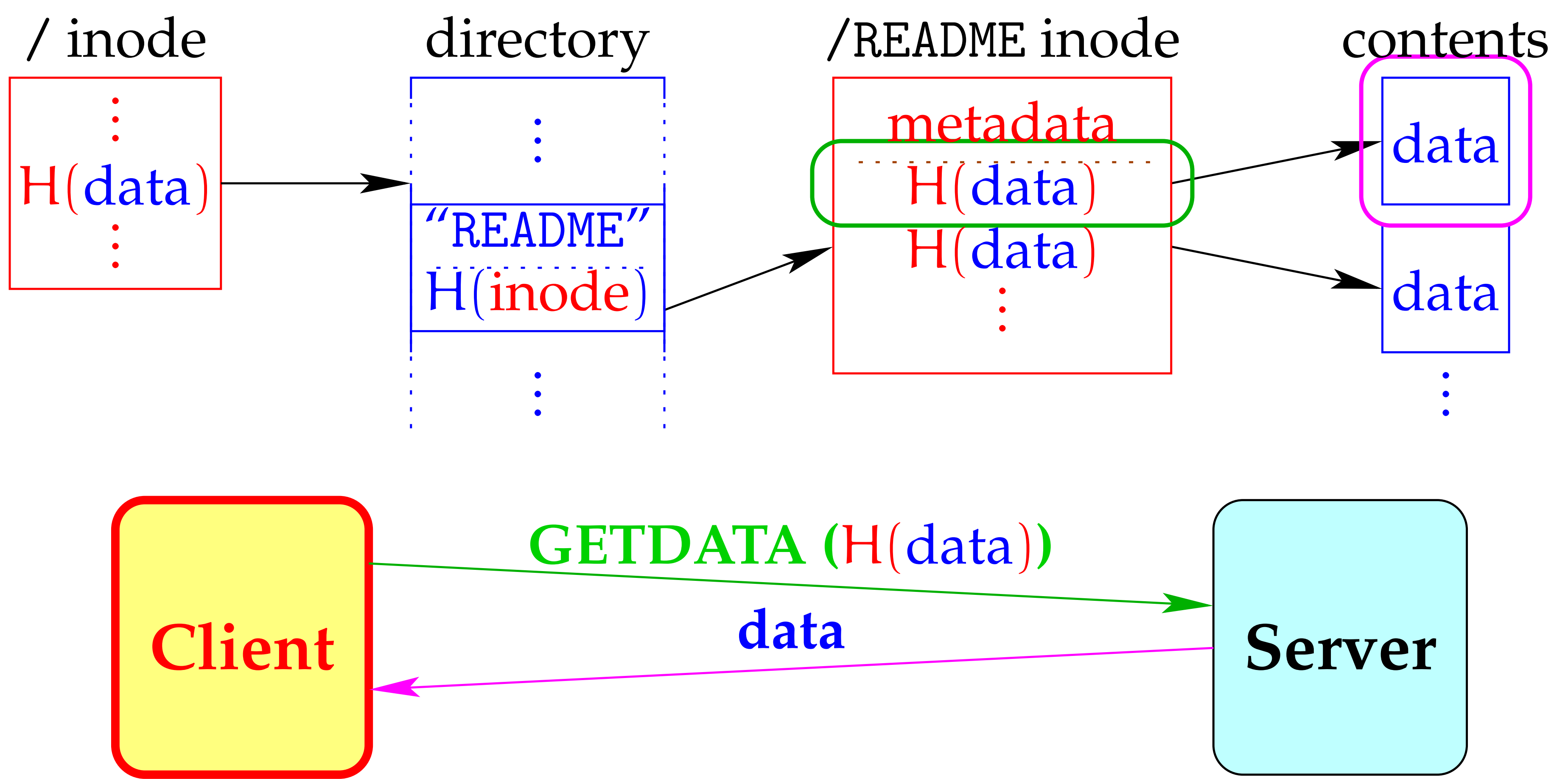
SFSRO Protocol



SFSRO Protocol

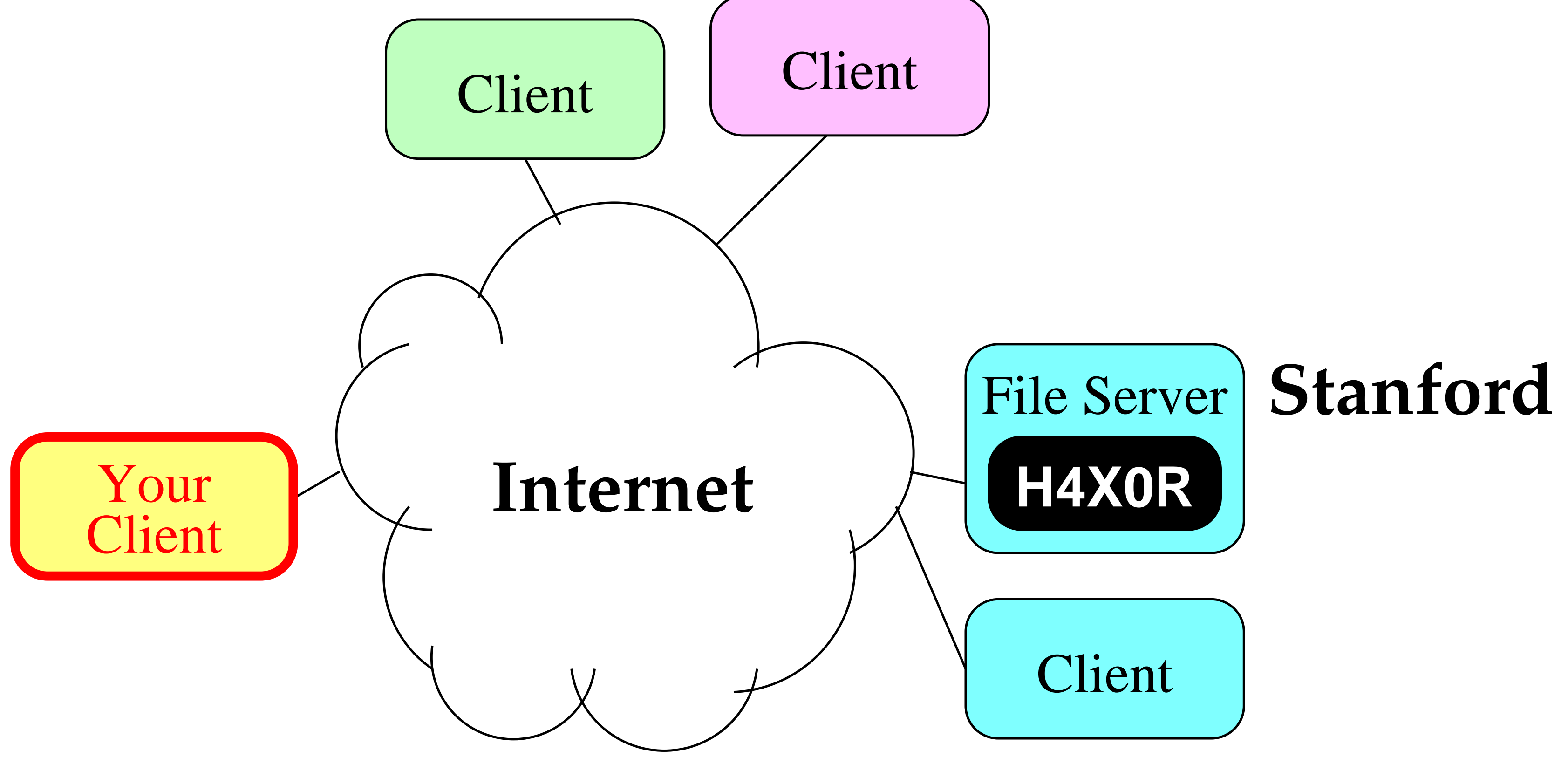


SFSRO Protocol



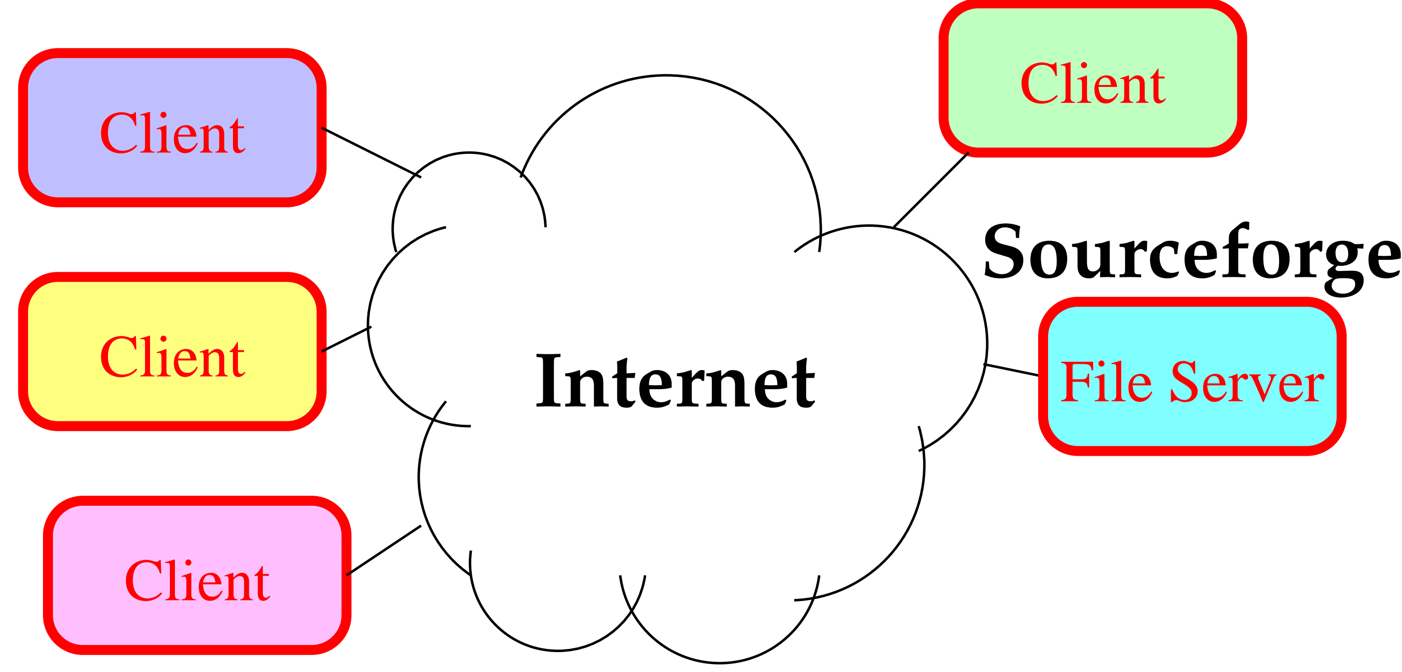
SUNDR

SUNDR: True end-to-end file system security



- **Normally trust file servers to return correct data**
  - Reject unauthorized requests, properly execute authorized ones
- **Should trust only clients of authorized users**
  - SUNDR can detect misbehavior *even if attacker controls server*

Motivation: Outsourcing data storage

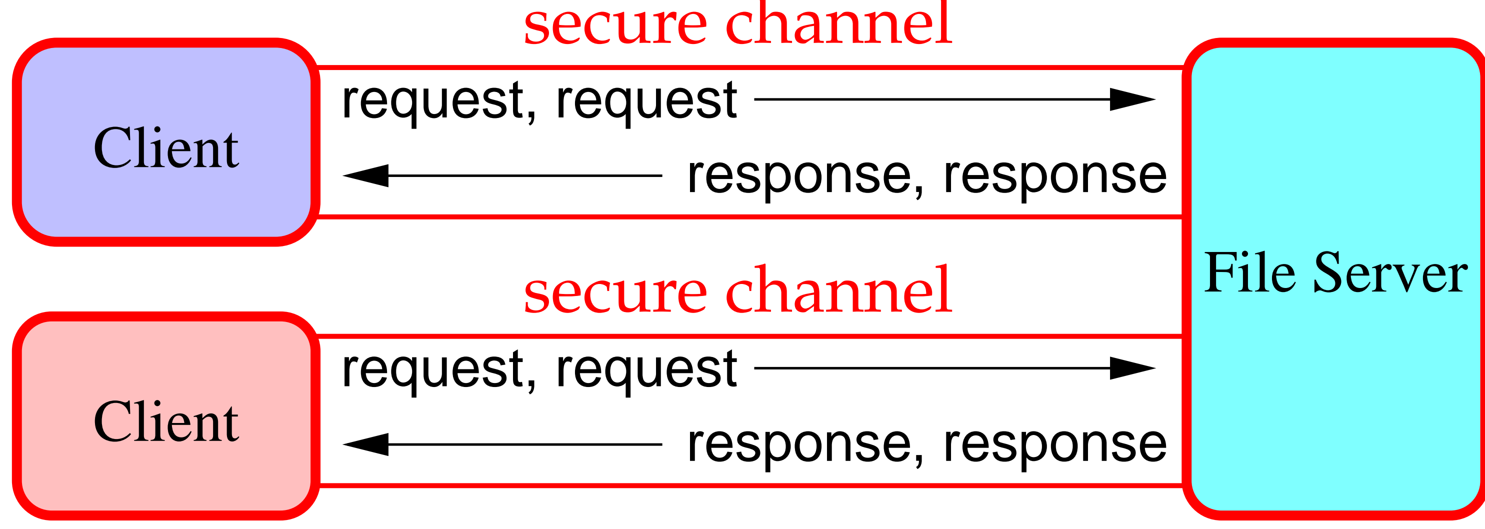


- E.g., Sourceforge hosting source repositories
- Attractive target of attack

A worrisome trend

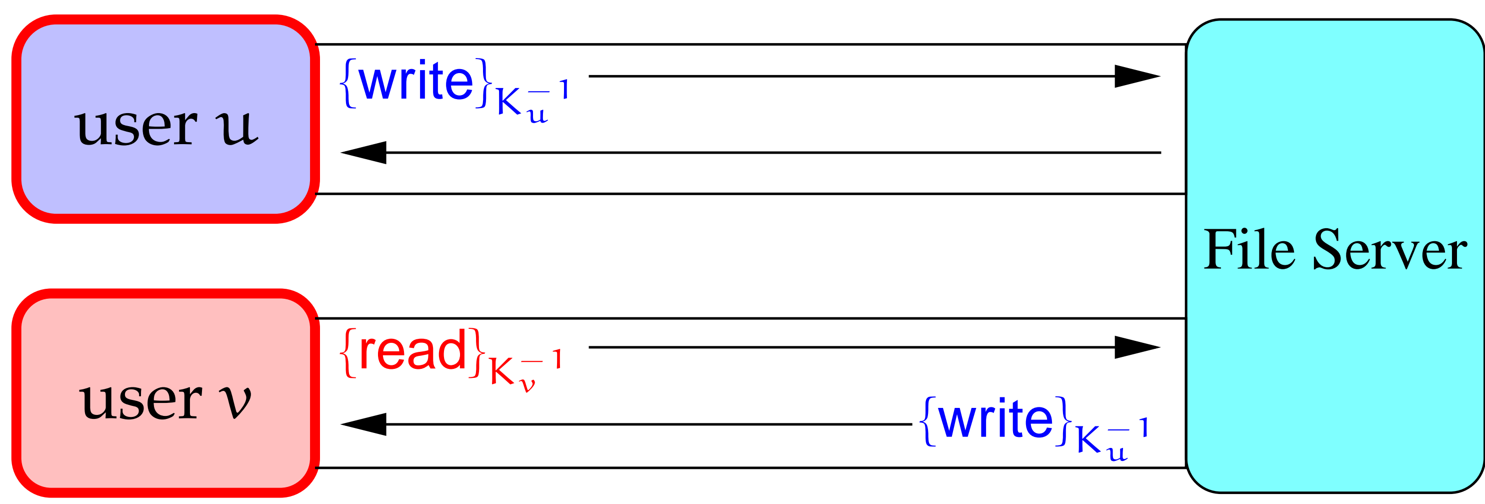
- **5/17/01: Apache development servers compromised**
  - Password captured by trojaned ssh binary at sourceforge
  - The integrity of all source code repositories is being individually verified by developers… - Apache press release
- **11/20/03: Debian administrators discover "root kit"**
  - at the 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



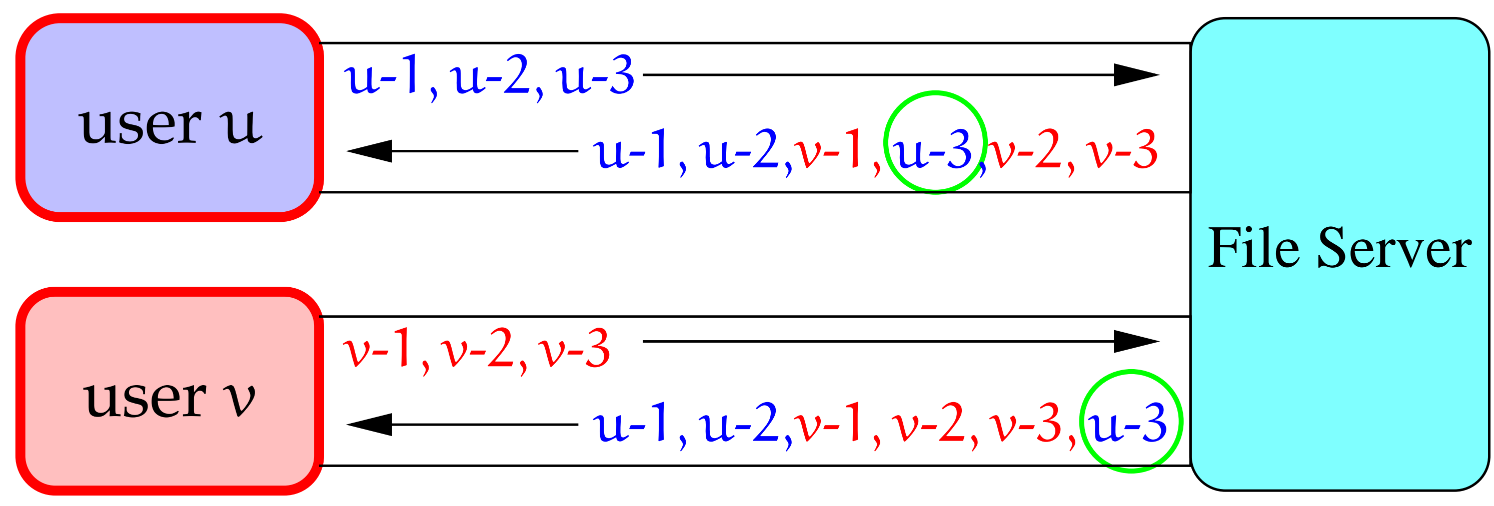
- **Clients & servers communicate over secure channels**
  - Network attackers can't tamper with requests
- **Server can't prove what requests it received**
  - Trust server to execute requests properly
  - Trust server to return correct responses

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

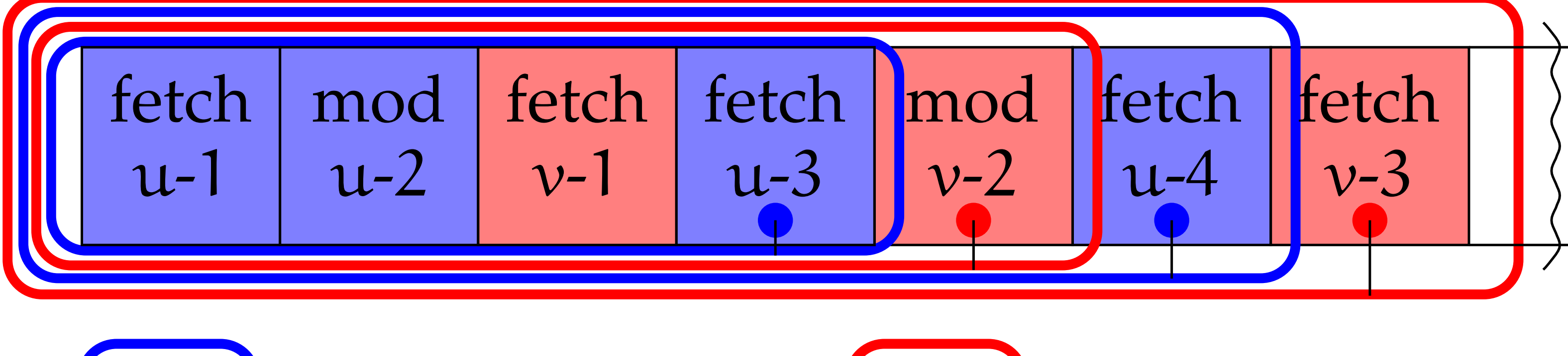
A Fetch-Modify interface

- **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 → fetch (dir & file), write+close → modify,
    truncate → modify, creat → 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
  - (Basically a formalization of "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



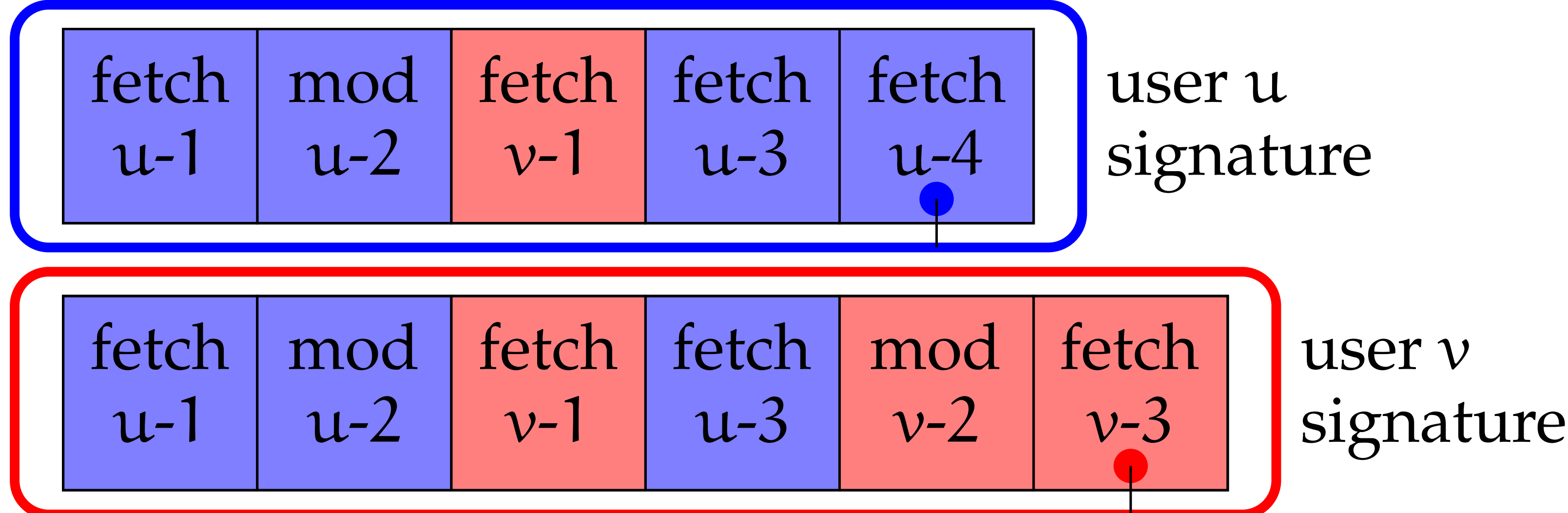
- **Detect reordering by signing entire FS history:**
- **PREPARE RPC – lock file system, download log**
  - Client checks signatures on log entries
  - Client checks that its previous operation is still in log
- **Client plays log to reconstruct FS state**
- **Client appends new operation, signs new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

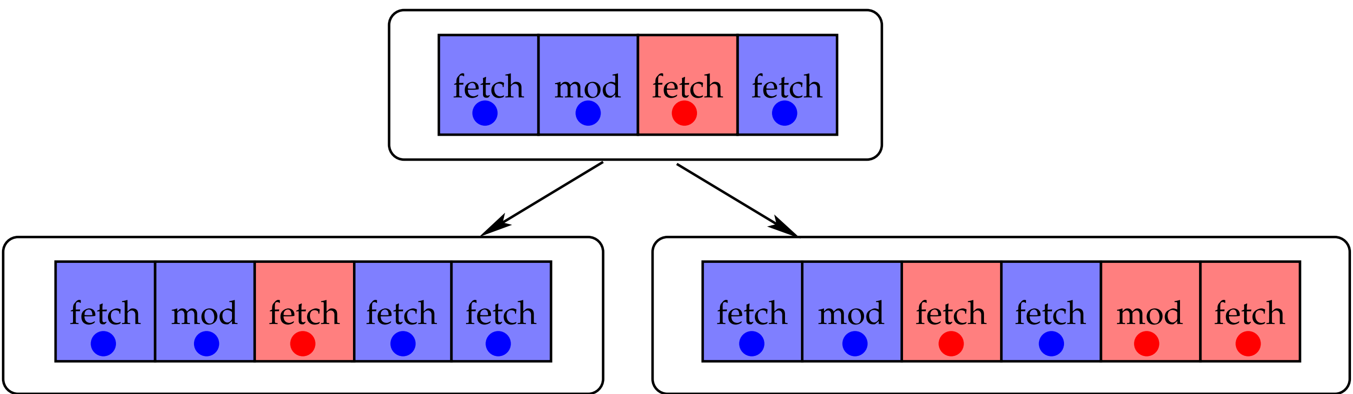
What can a malicious server do?

- **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

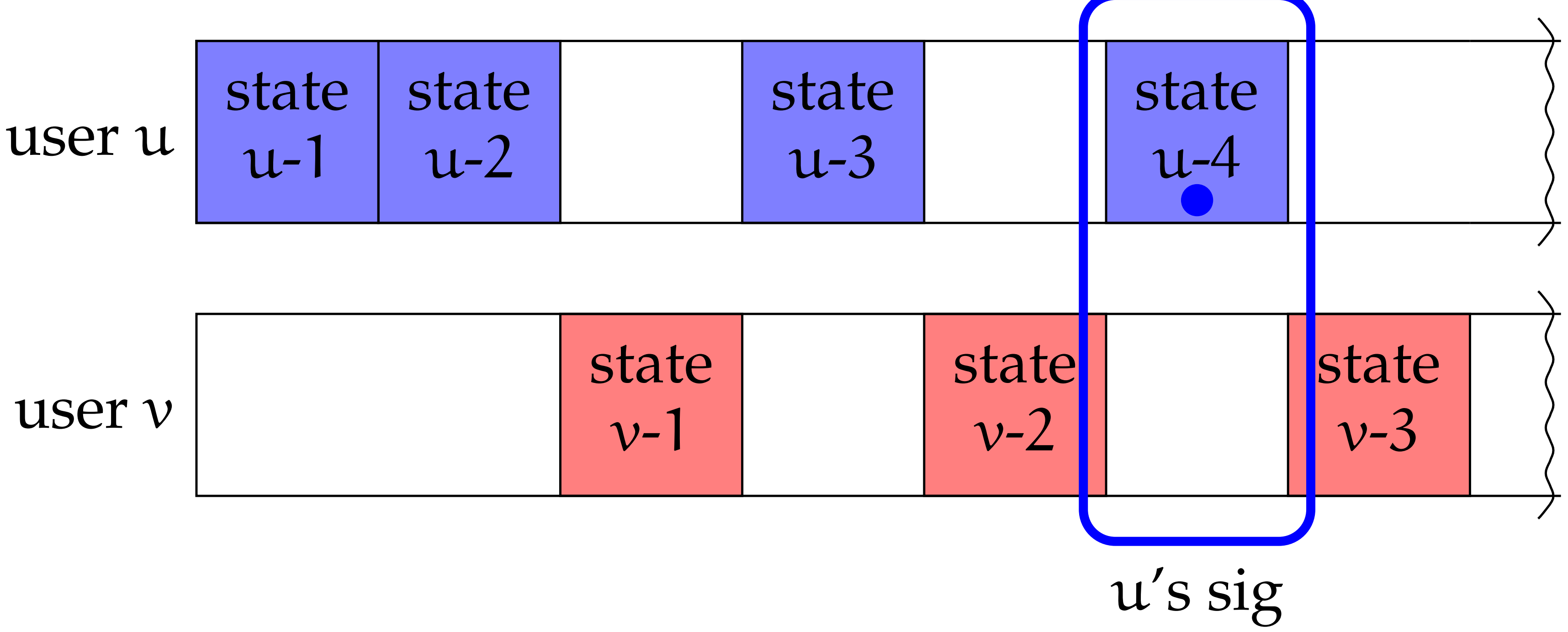
Implications of fork consistency

- **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?

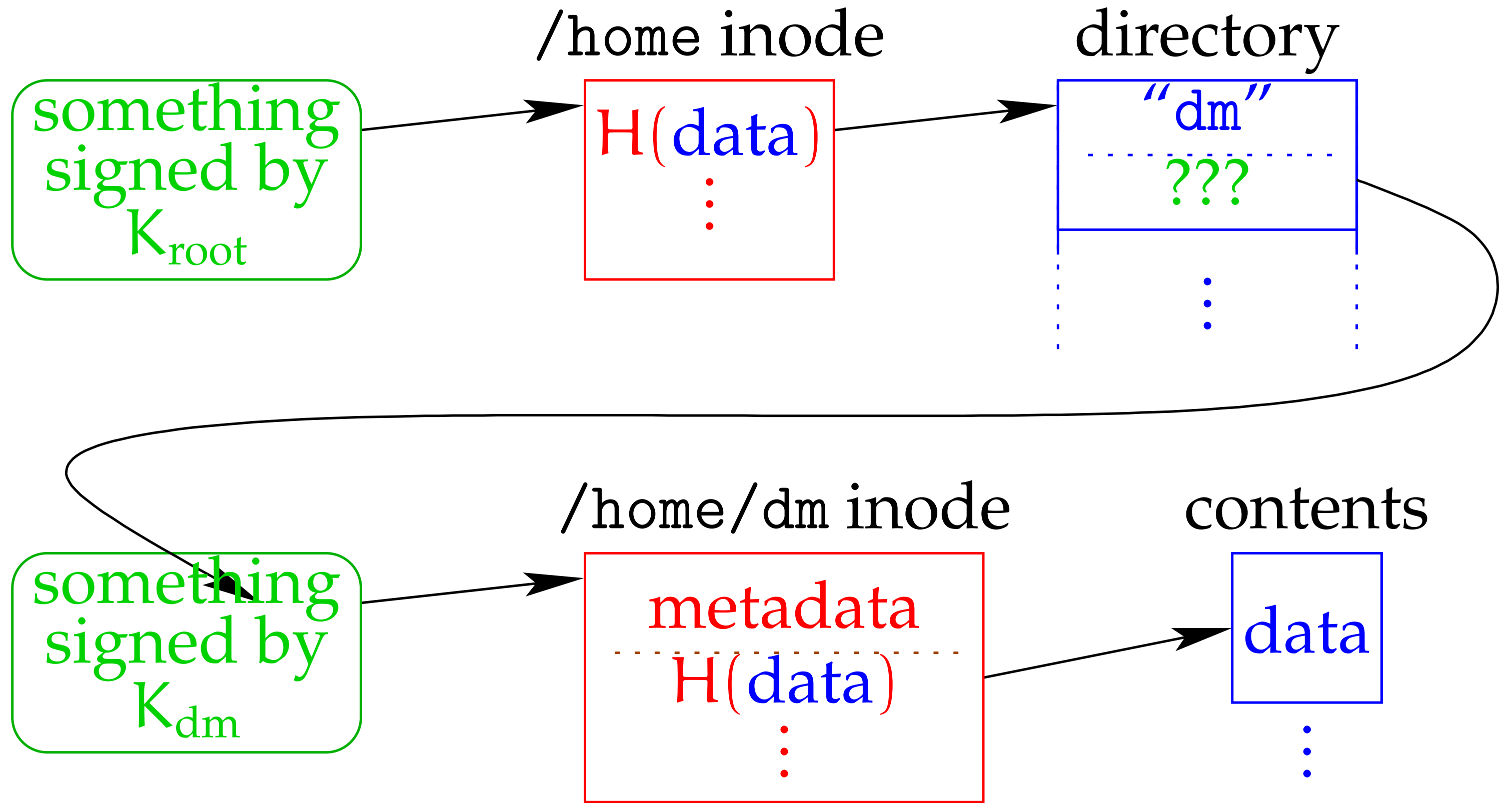
A plan for signing snapshots



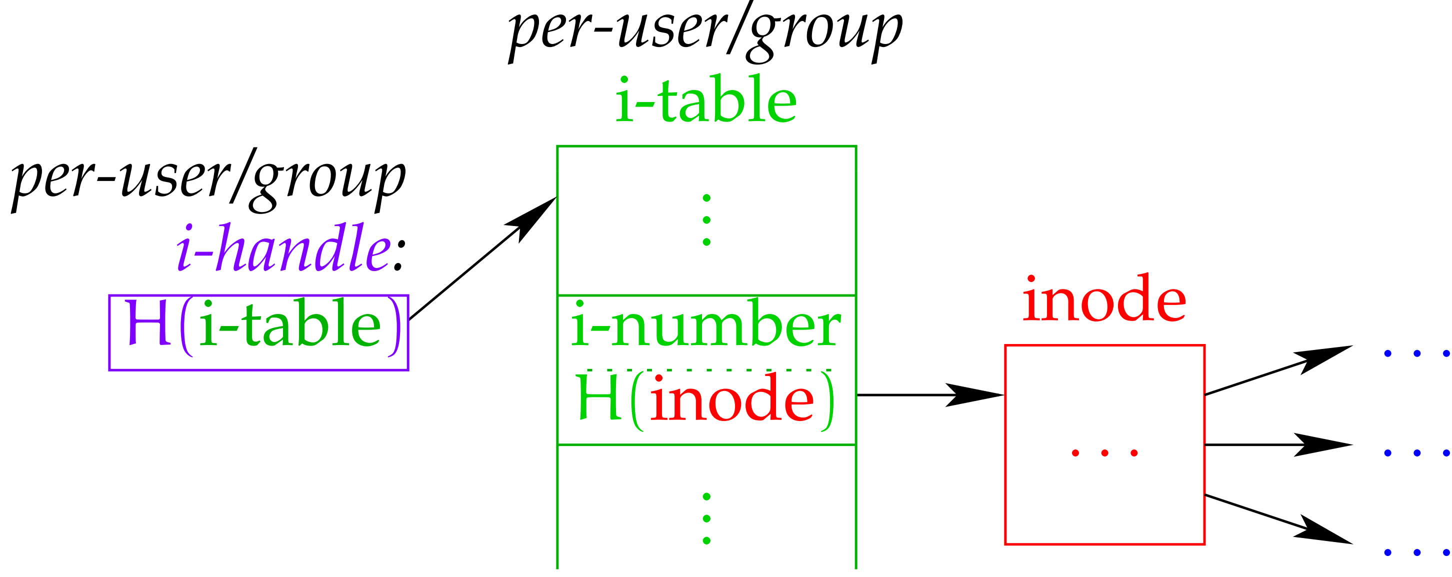
- Somehow represent snapshots of each user's files in a way that they can be combined…
- Somehow prevent re-ordering of users' snapshots…

Combining snapshots

- **A user's directory might contain another user's file**
  - E.g., root owns `/home`, dm owns `/home/dm`
  - dm needs to update file w/o having root re-sign anything
  - root must sign name "`/home/dm`" while dm signs contents

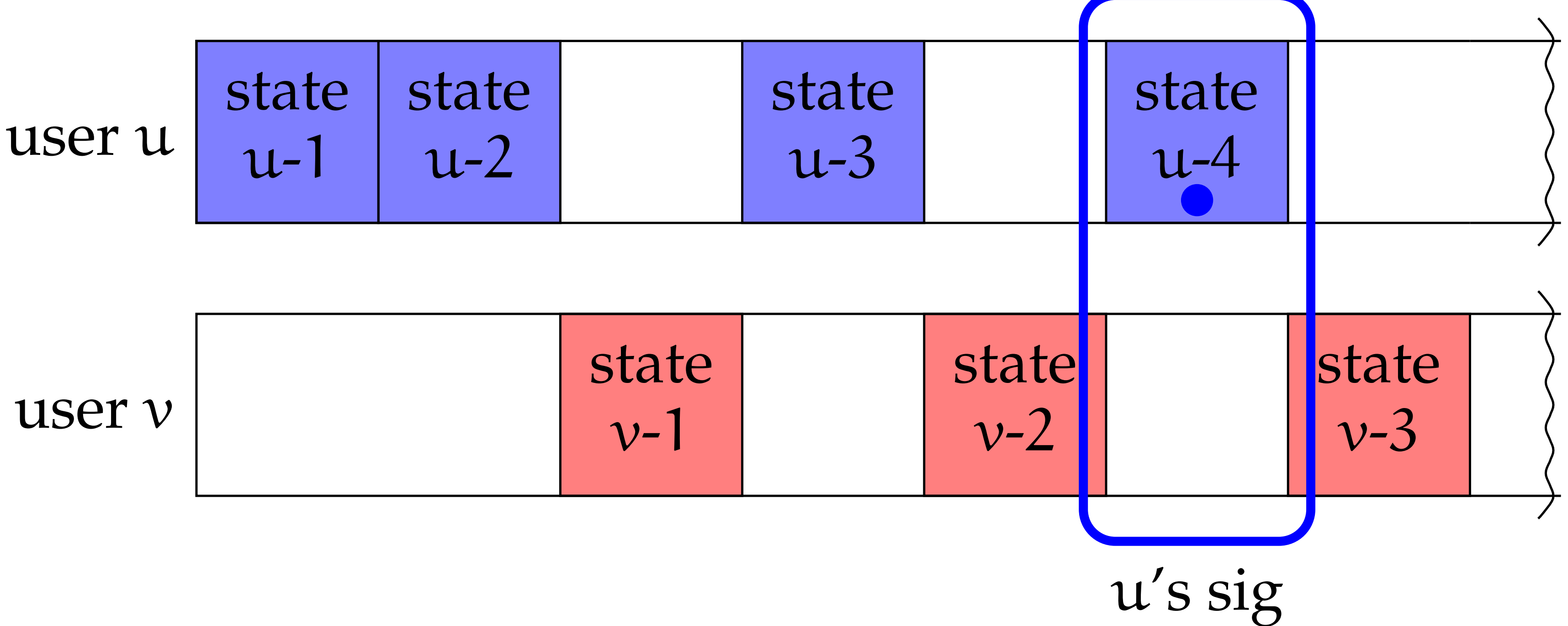


Per-user or -group i-numbers



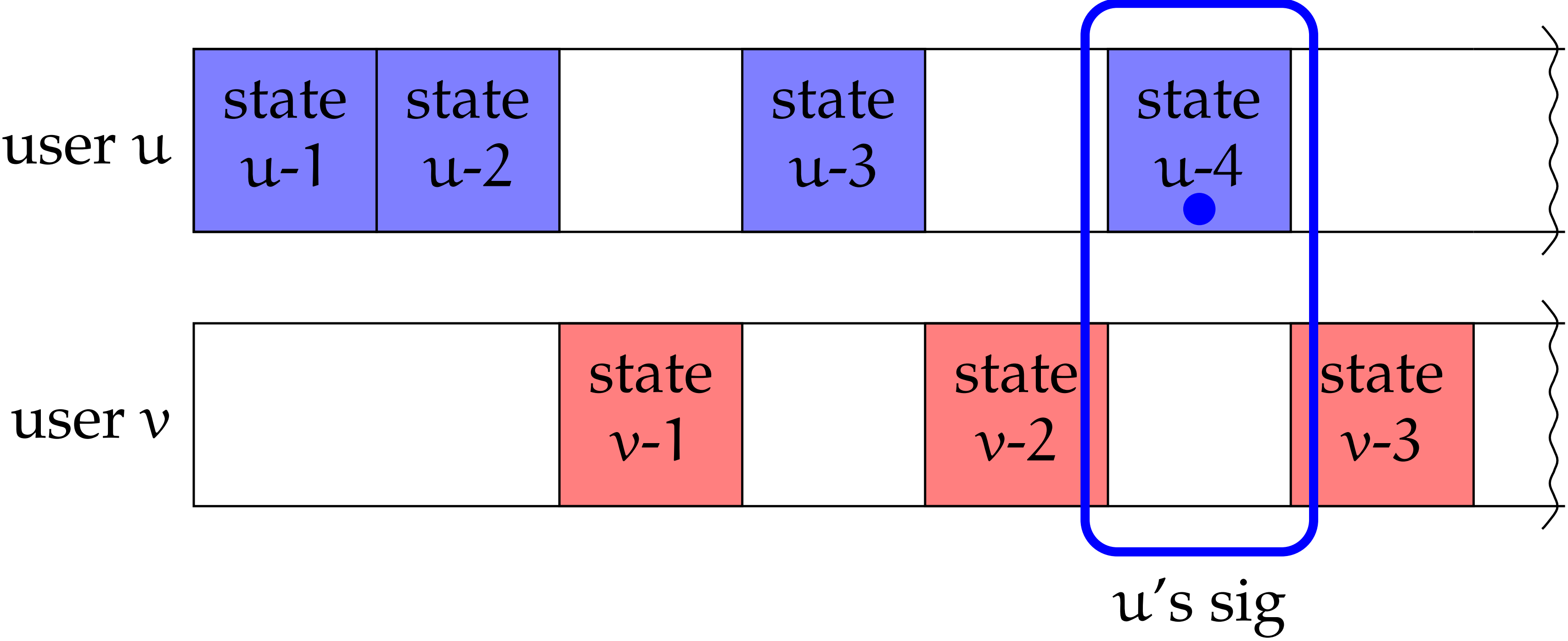
- **Add a level of indirection to SFSRO data structures**
- **SUNDR directory entry:** file name / ⟨user/group, i-number⟩
- **Per-user/group *i-tables* map** i-number → H(inode)
- **Hash each** i-table **to a short *i-handle* users can sign**

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:**

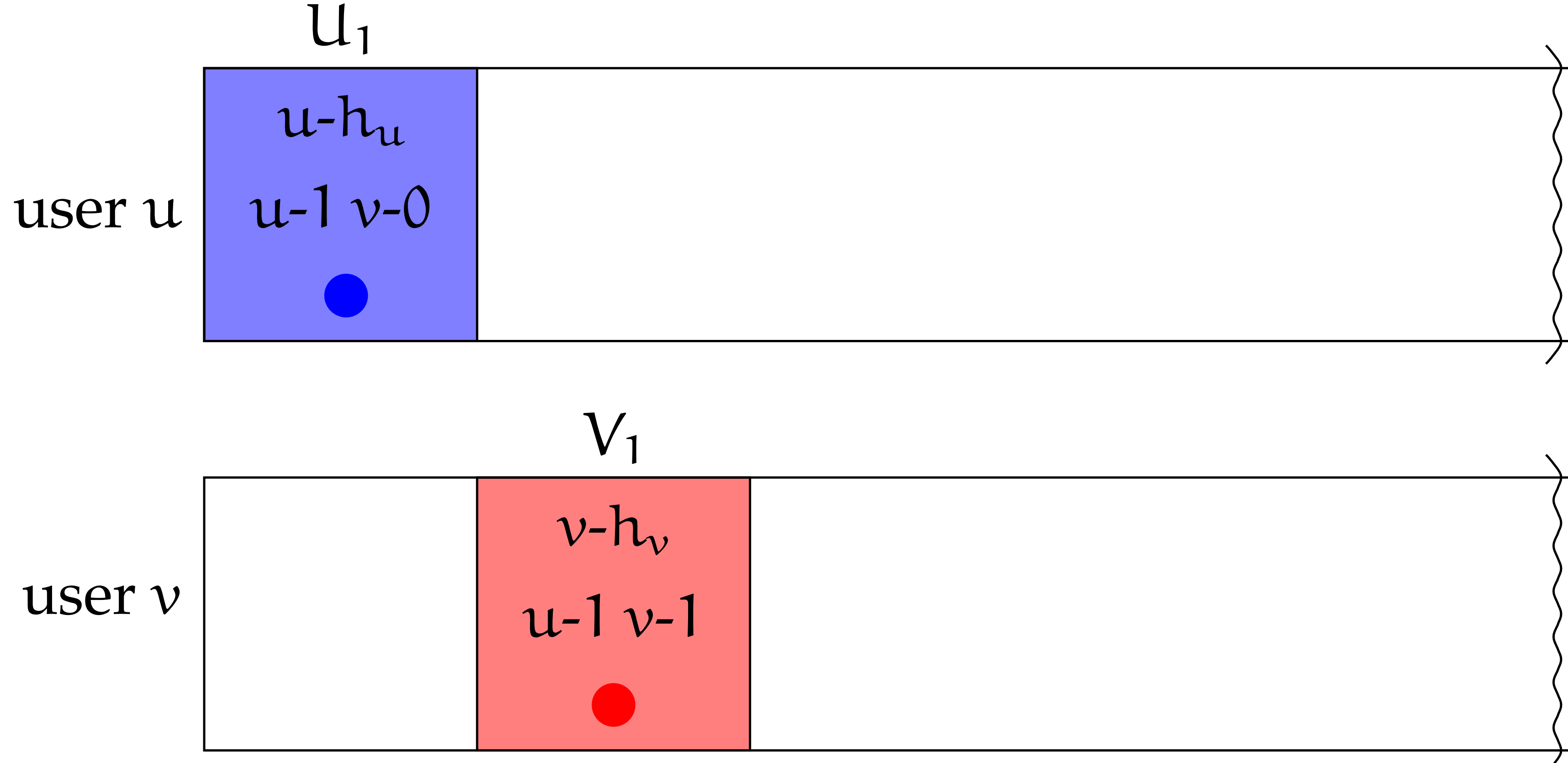
$$\textit{version structure:}\ \{\ \underbrace{u\text{-}h_u}_{\text{i-handle}}\ ,\ \underbrace{u\text{-}4\ v\text{-}2}_{\text{version vector}}\ \}_{K_u^{-1}}$$

- **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

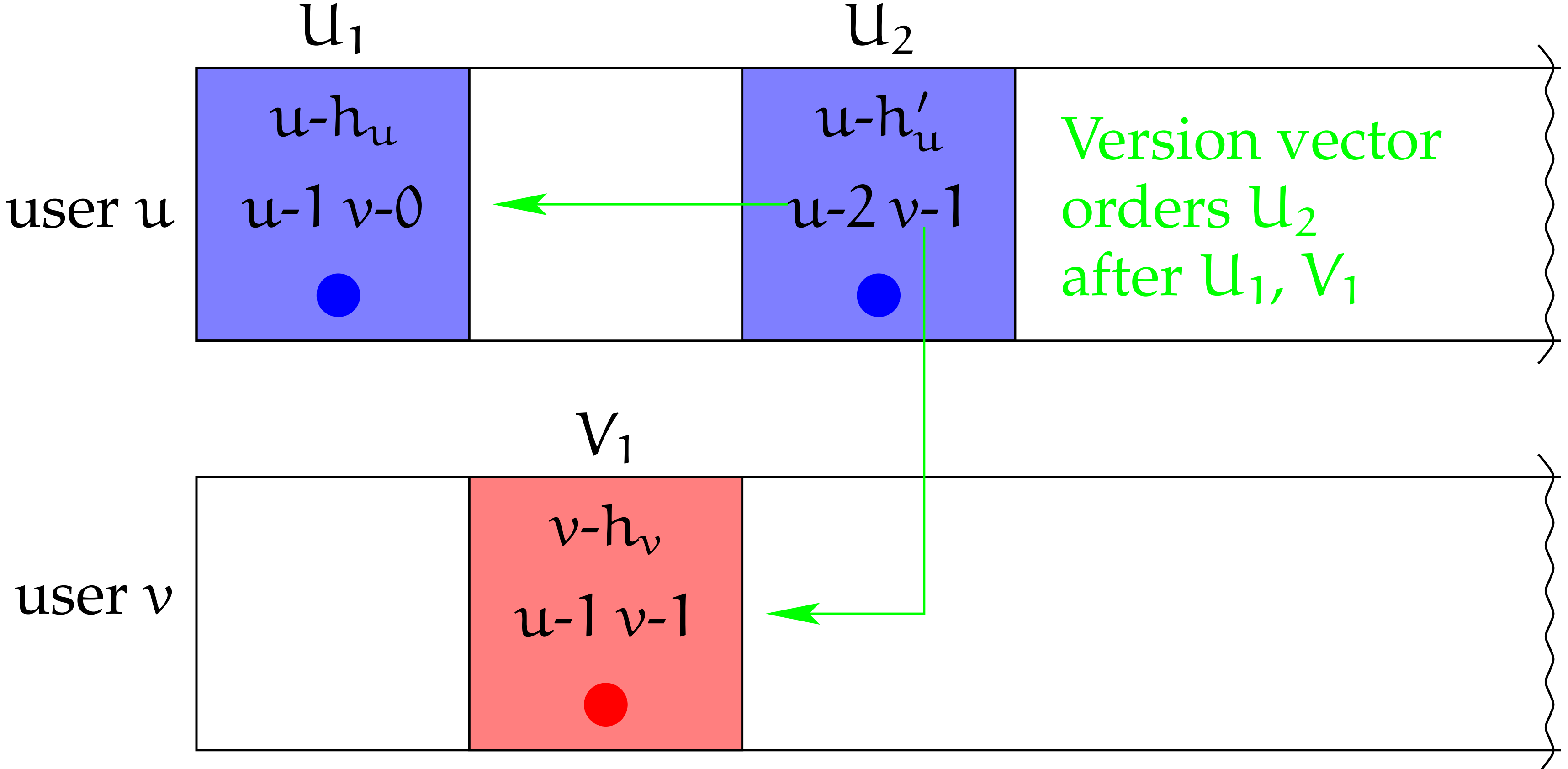
- **Still no concurrent updates**
- **Server maintains *version structure list* or VSL**
  - Contains latest version structure for each user/group
- **To fetch or modify a file, u's client makes 2 RPCs:**
  - PREPARE: Locks FS, returns VSL
  - Client sanity-checks VSL (ensures it is totally ordered)
  - 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



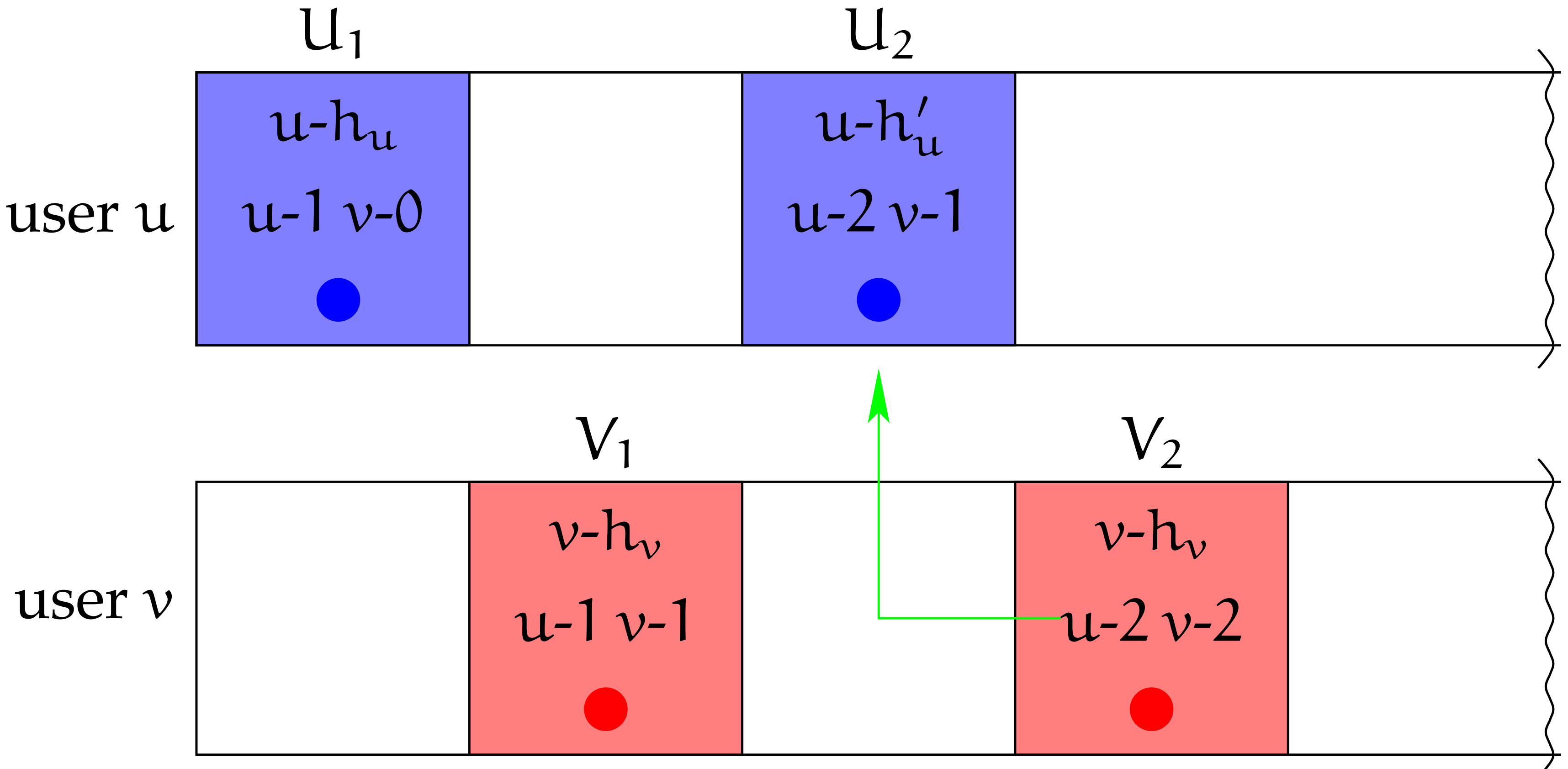
- **Users u and v each start at version 1 (sign U_1 & V_1)**

Example: Honest server



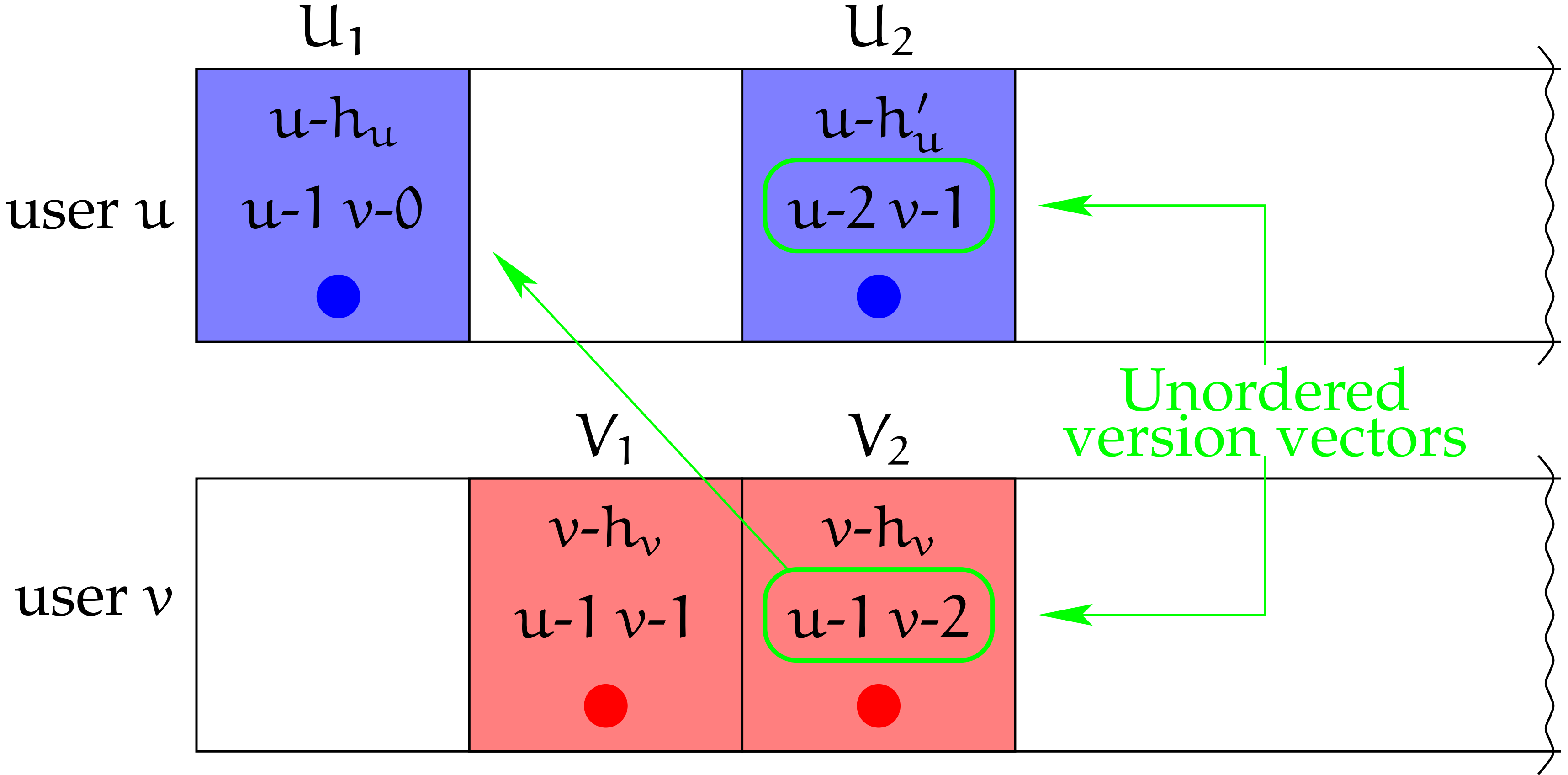
- **Users u and v each start at version 1 (sign U_1 & V_1)**
- **u modifies file f, signs U_2 w. new i-handle h'_u**

Example: Honest server



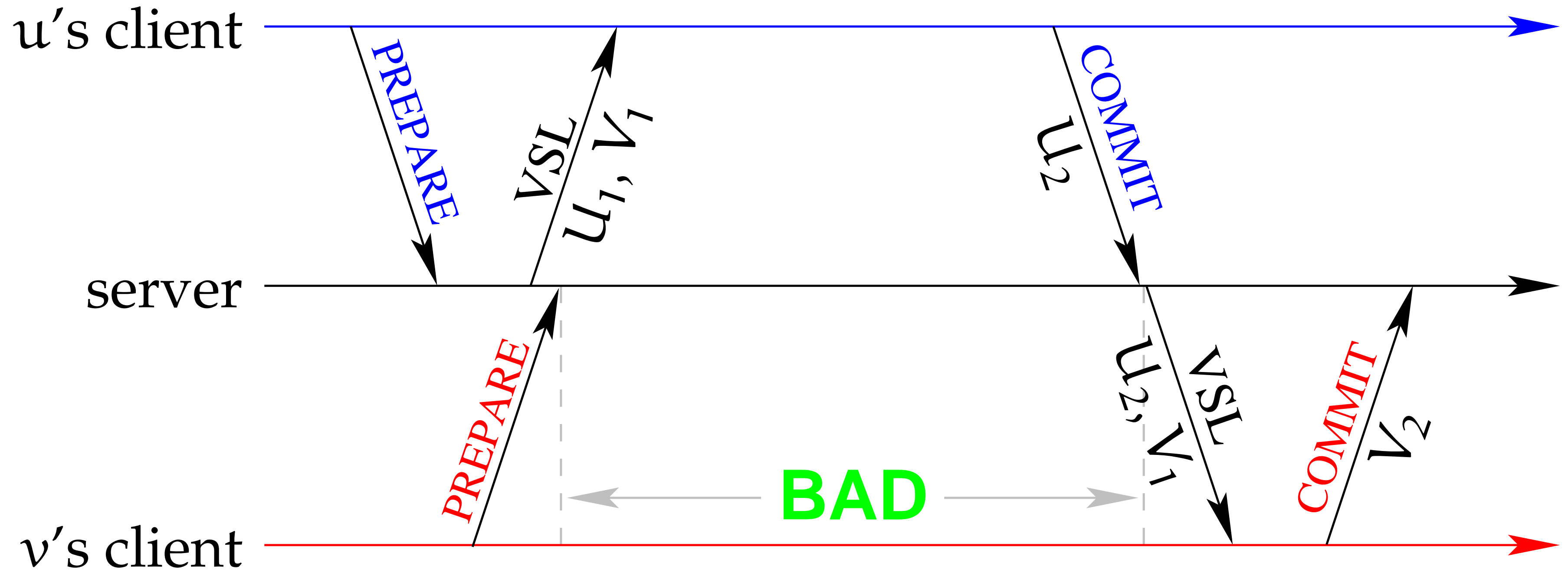
- **Users u and v each start at version 1 (sign U_1 & V_1)**
- **u modifies file f, signs U_2 w. new i-handle h'_u**
- **v fetches f, signs V_2 which reflects having seen U_2**

Example: Malicious server



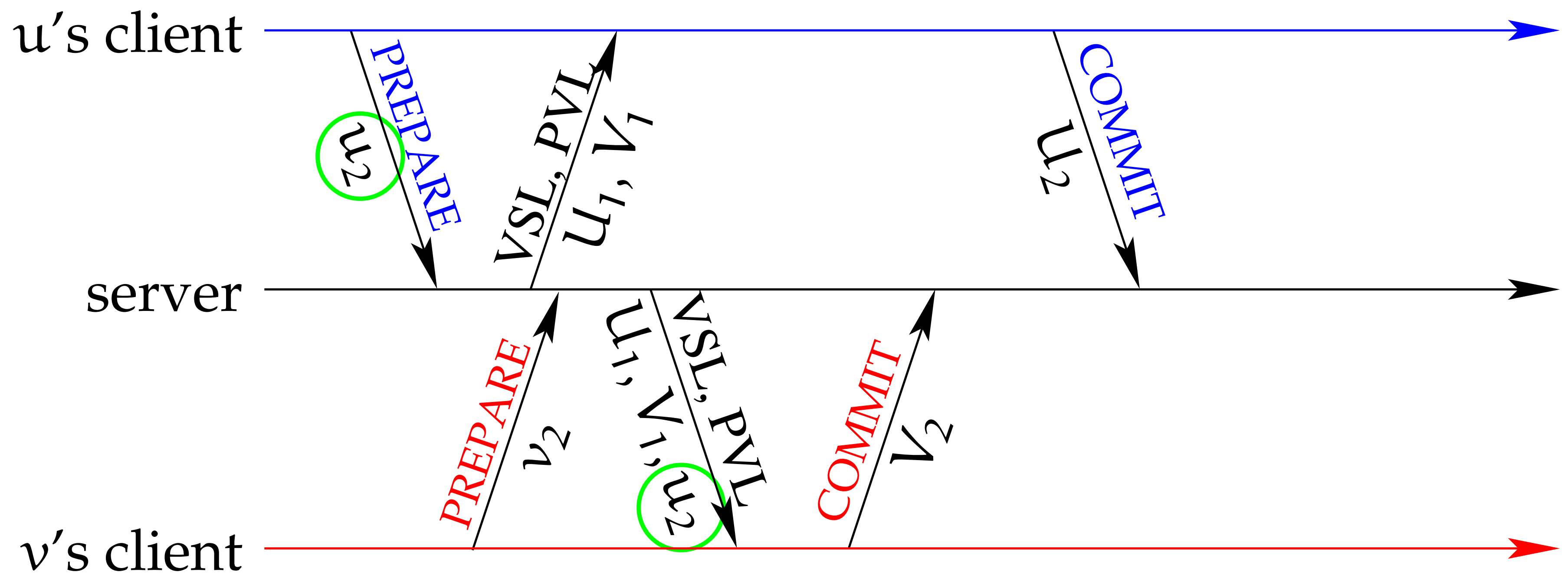
- **Suppose server hadn't shown u's modification of f to v**
- **Now $U_2 \not\leq V_2$ and $V_2 \not\leq U_2$**
  - u or v will detect attack upon seeing any future op by other

Limitations of serialized SUNDR



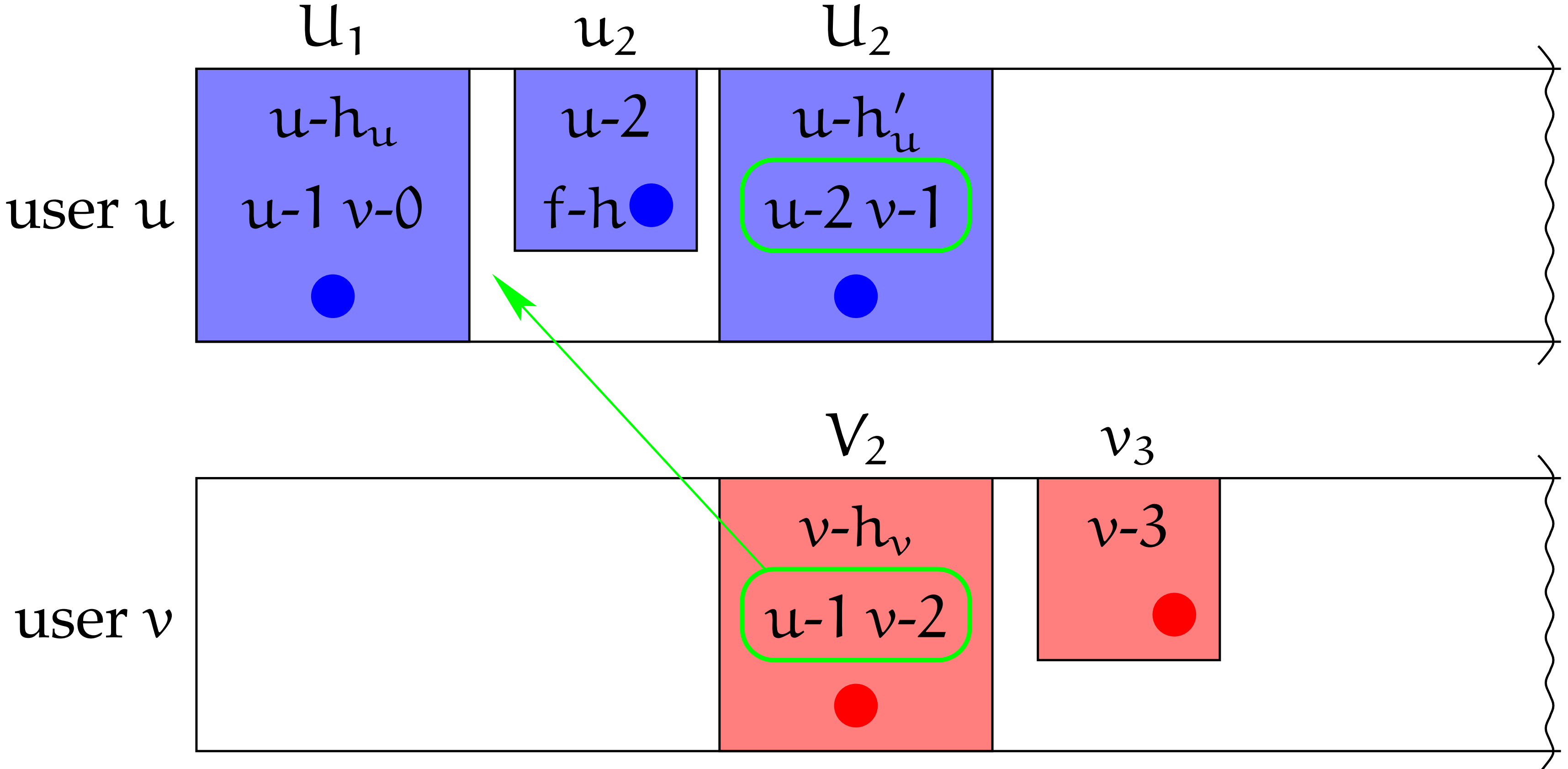
- **Honest server can only allow one operation at a time**
  - E.g., server must send u_2 to v to prevent fork on last slide
  - Must wait *even if* v_2 *doesn't observe any changes made in* u_2
- **Without concurrency, get terrible I/O throughput**

Solution 3: SUNDR



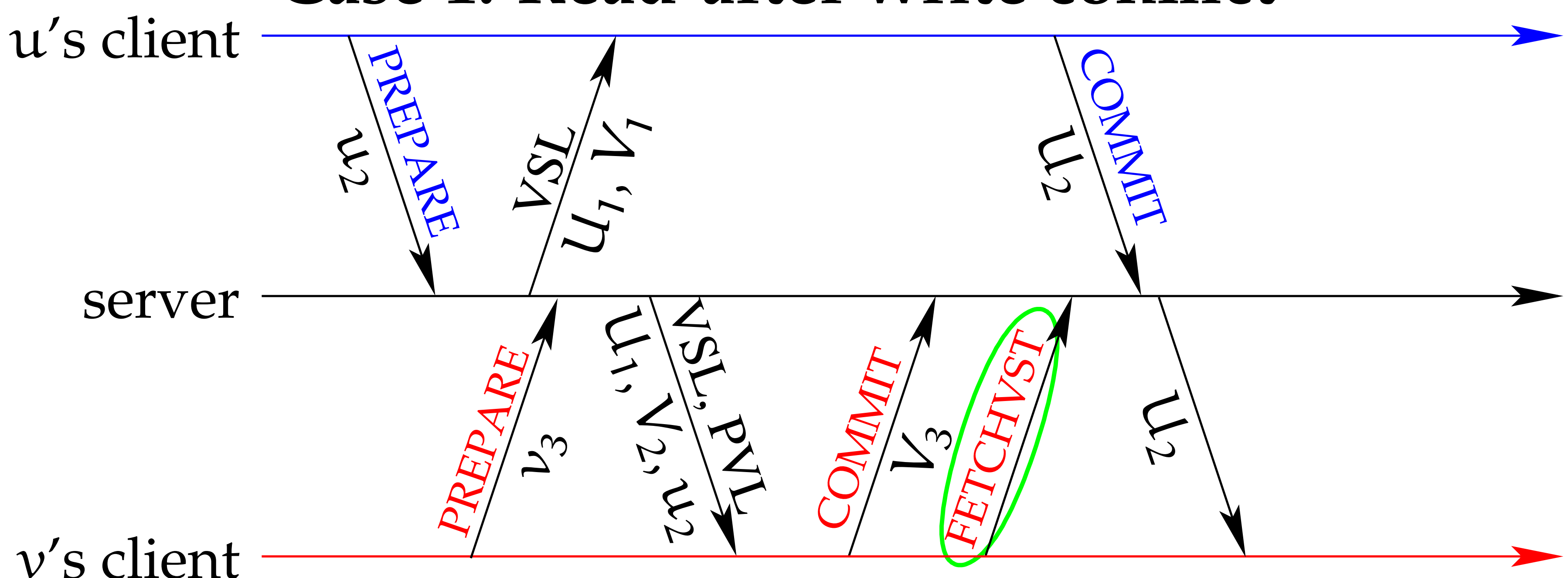
- **Pre-declare operations in signed *update certificates***
  - $u_2 = \{$"In vstruct U_2, I intend to change file f to hash h."$\}_{K_u^{-1}}$
- **Server keeps uncommitted update certificates in *Pending Version List* or PVL, returns with VSL**
- **Plan: Have v compute V_2 w/o seeing U_2 if it sees 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

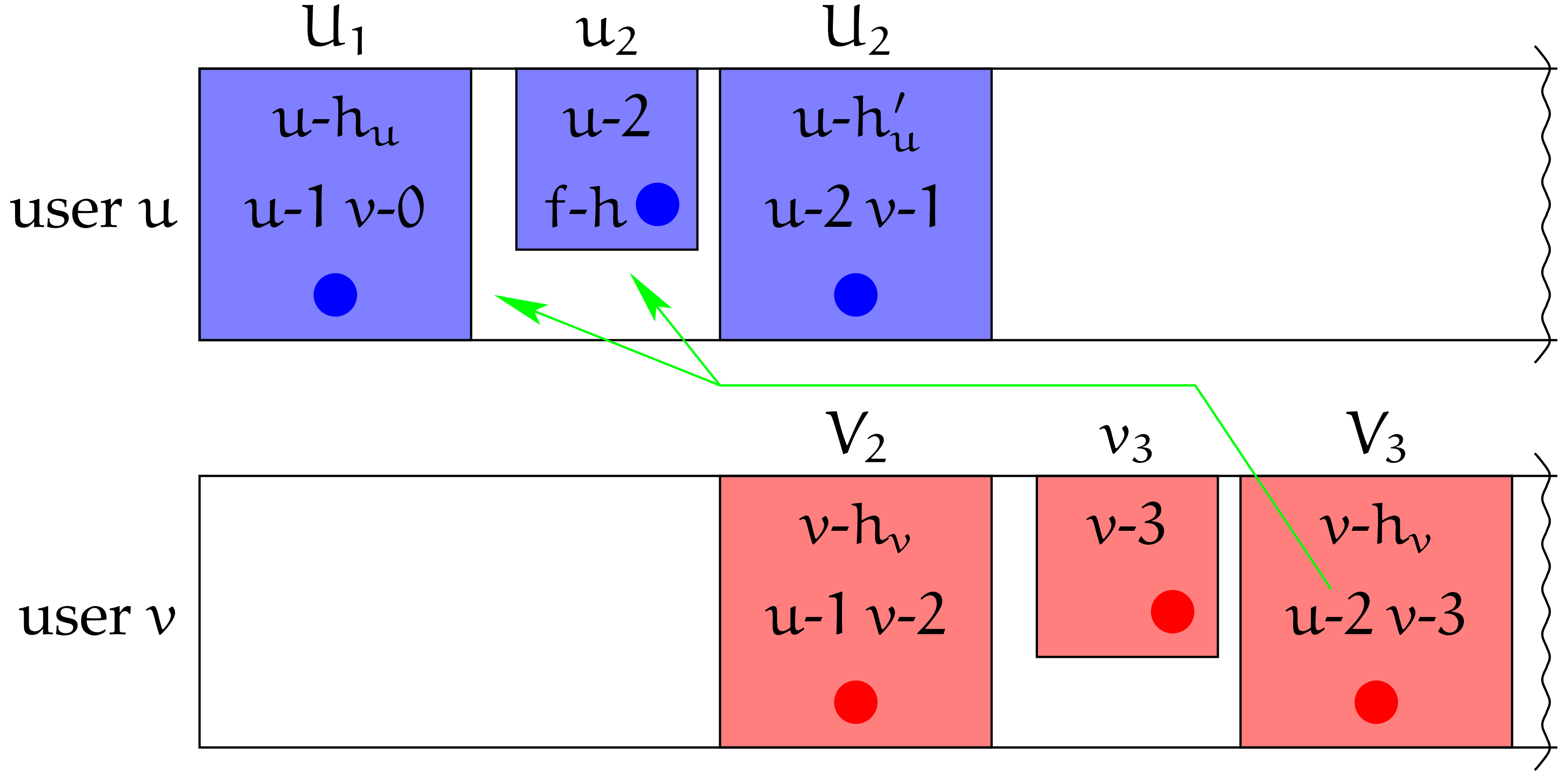


- **Must *not* show effects of u_2 to v's application**
  - Recall: when v sees change by u, should guarantee no attack
- **Solution: Wait for vstruct w. new FETCHVST RPC**
  - Example:

$$U_2 = \{u\text{-}2\ v\text{-}1\} \quad V_2 = \{u\text{-}1\ v\text{-}2\}$$

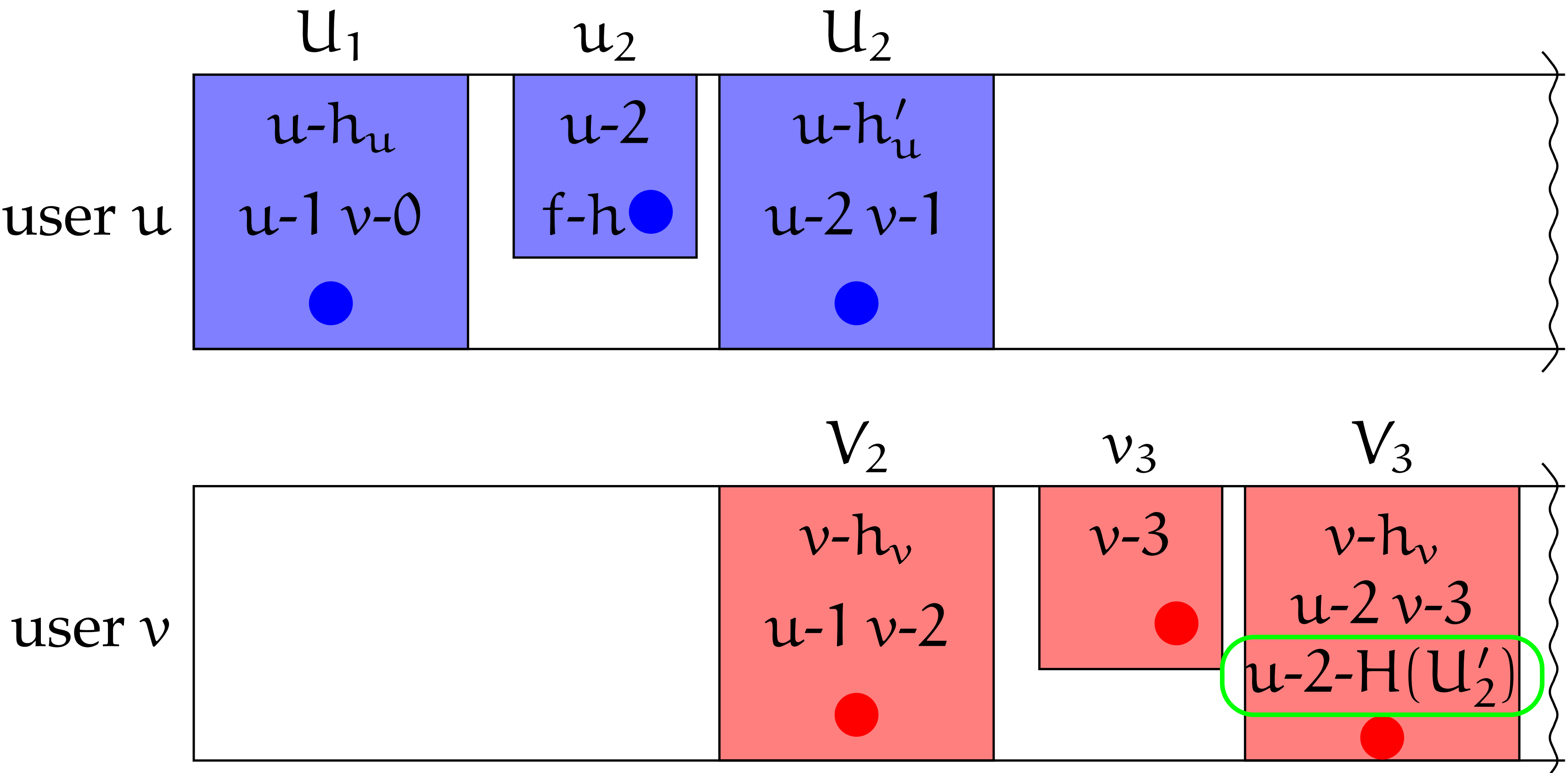
v detects attack as $U_2 \not\leq V_2$ (in VSL) and $V_2 \not\leq U_2$

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

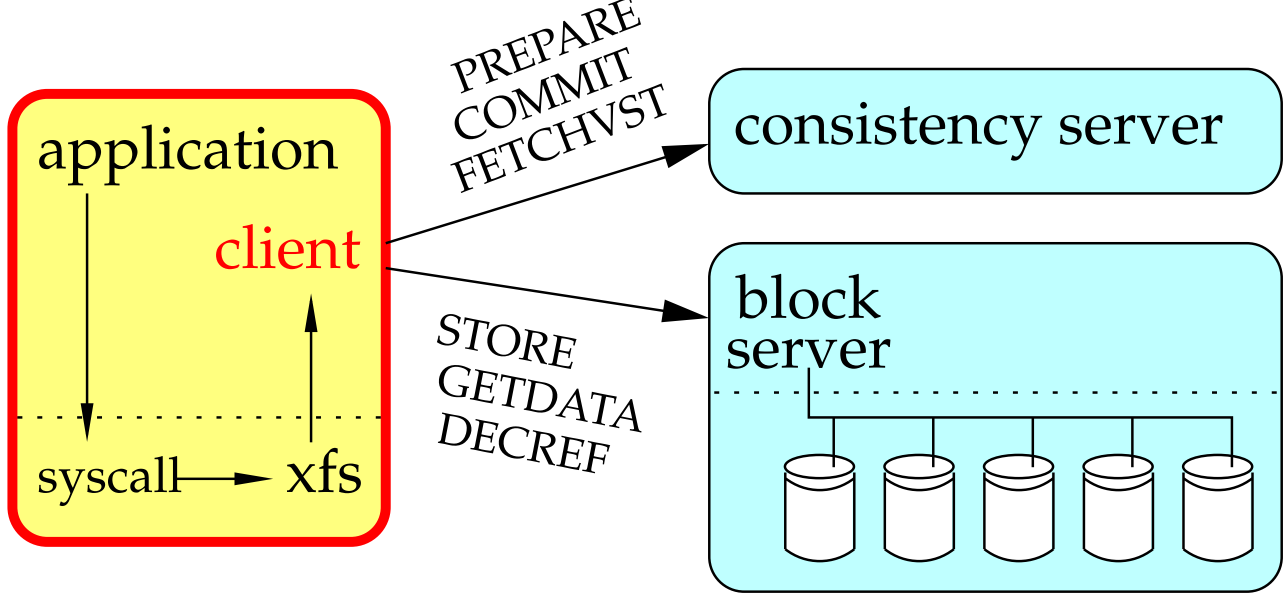


- **Vstruct includes hashes of other anticipated vstructs**
  - Omit i-handles so contents deterministic given order of PVL
- **Redefine $\leq$ to require that hashes match**
  - E.g., $U_2 \not\leq V_3$, because V_3 contains hash of $U'_2 = \{u\text{-}2\ v\text{-}2\} \neq U_2$

Summary of SUNDR properties

- **Looks like a file system**
  - E.g., could use for CVS access to sourceforge
- **Only two ways for server to subvert integrity**
  - Can fork users' views of file system (recover like Ficus)
  - Can throw away your data (recover from backup and/or untrusted clients' caches)
- **Concurrent operations from different clients**

Implementation

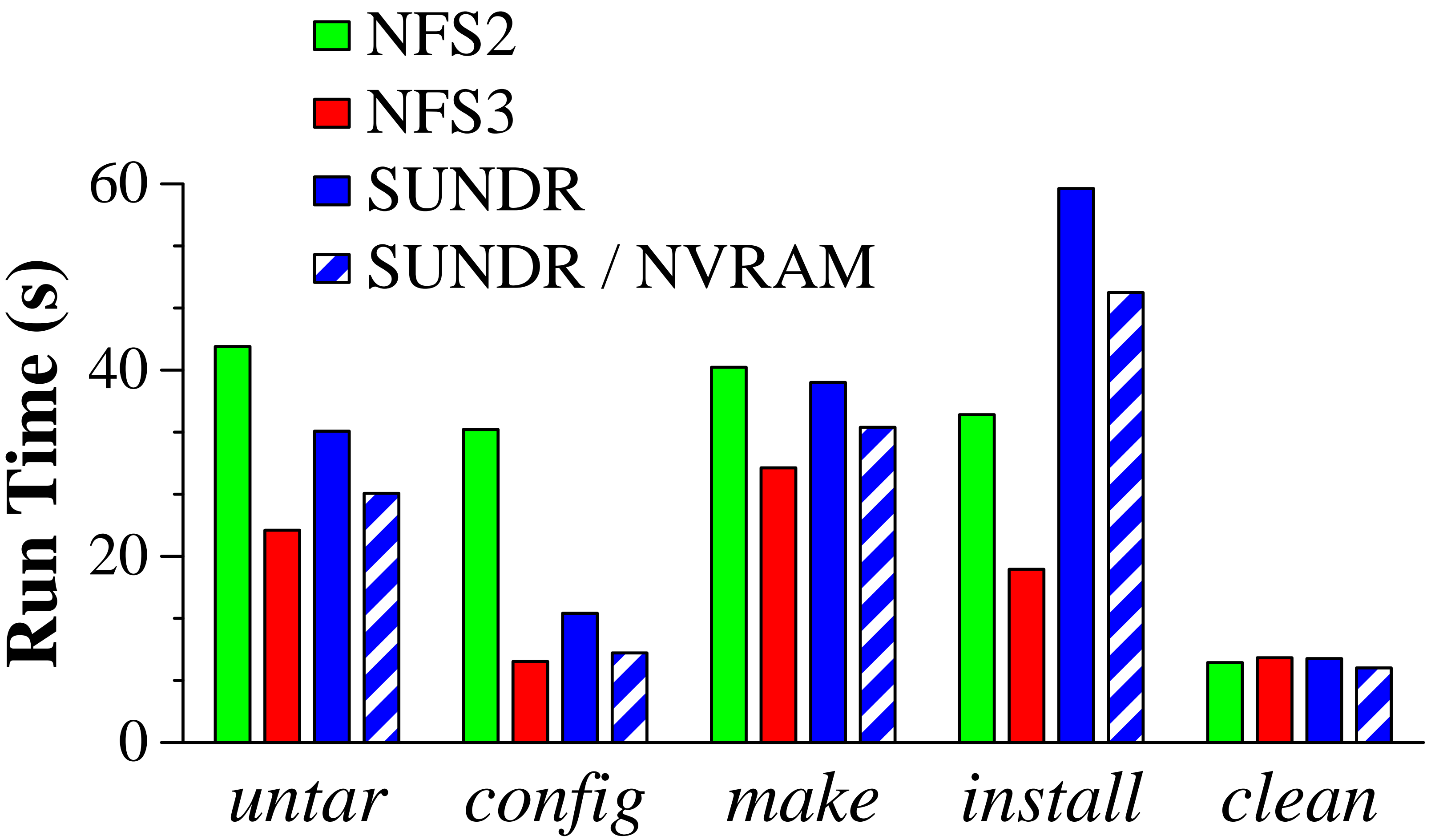


- **Client based on xfs device driver**
  - xfs part of Arla, a free AFS implementation
  - Designed for AFS-like semantics
- **Server split into two daemons**
  - *Consistency server* handles update certs, version structs
  - *Block server* stores bulk of data
  - Can run on same or different machines

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

Related work

- **Byzantine Fault Tolerance**
  - File systems using BFT: BFS, Farsite, OceanStore/Pond
  - With 4 replicas, tolerate 1 compromise
- **Ordering of events**
  - Linearizability, version vectors, timeline entanglement, Smith-Tygar/Reiter-Gong
- **Merkle trees**
  - Merkle signatures, Duchamp, BFS, TDB, CFS [Dabek], PFS, Venti
- **Cryptographic storage**
  - Swallow, CFS [Blaze], PFS, Sirius, Plutus, Miller

Conclusions

- **Don't "lock down" major infrastructure with fences**
  - Hard to do uniformly securely for a large infrastructure
  - Fences make systems painful to use, impede innovation
- **Instead, take the end-to-end approach to security**
  - Don't be afraid to redefine your security properties
  - Eliminate trust w. novel applications of cryptography
- **Three examples of this approach:**
  - SFS: Shrink fence to exclude key management
  - SFSRO: Protect an essentially unfenceable system
  - SUNDR: New notion of consistency allows vastly less trust

Stanford Secure Computer Systems Group

http://www.scs.stanford.edu/

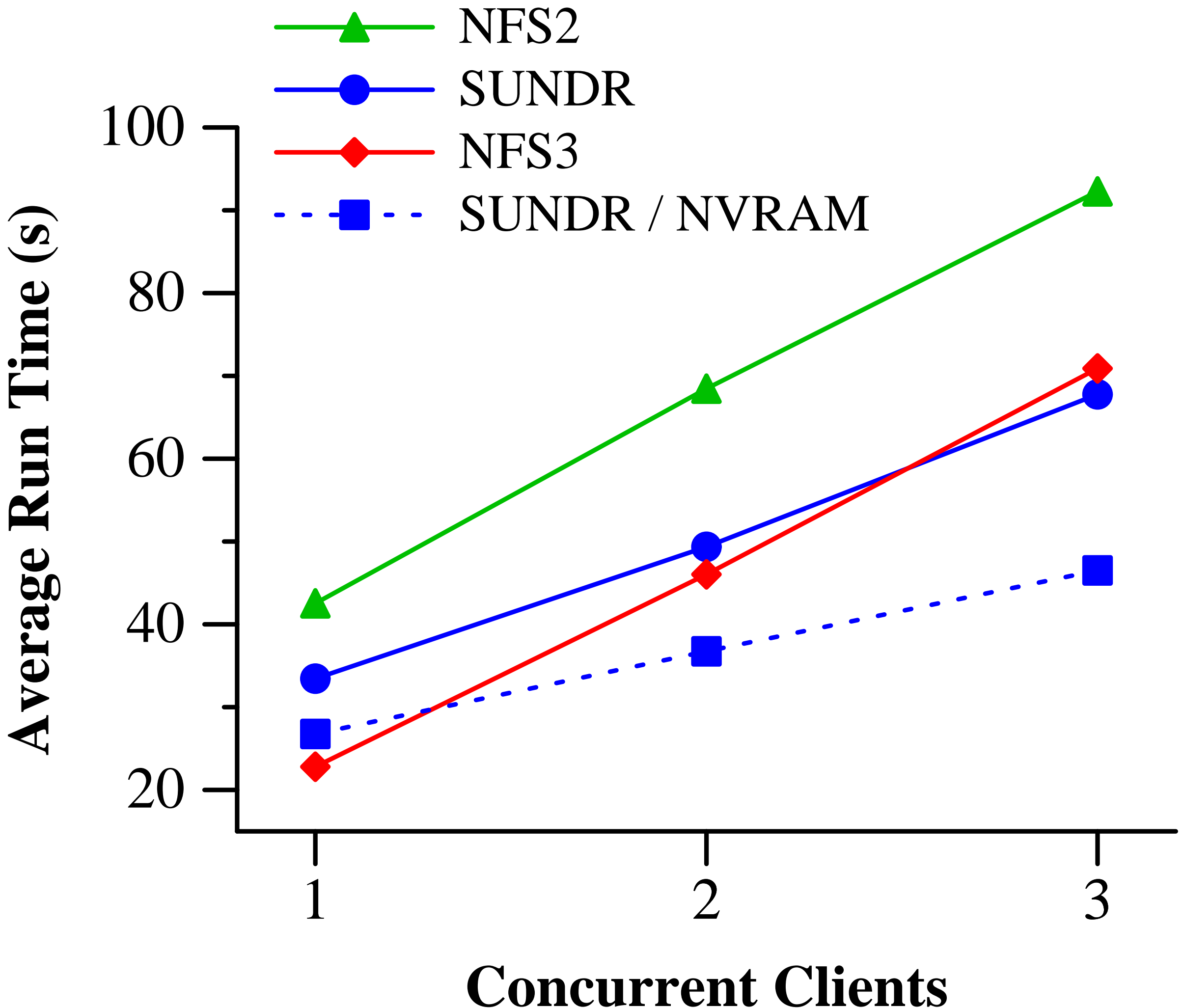
Recovery

- **Only two kinds of attack to recover from**
  - Forking attack (previously addressed)
  - Server throwing away data
- **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

Malicious users

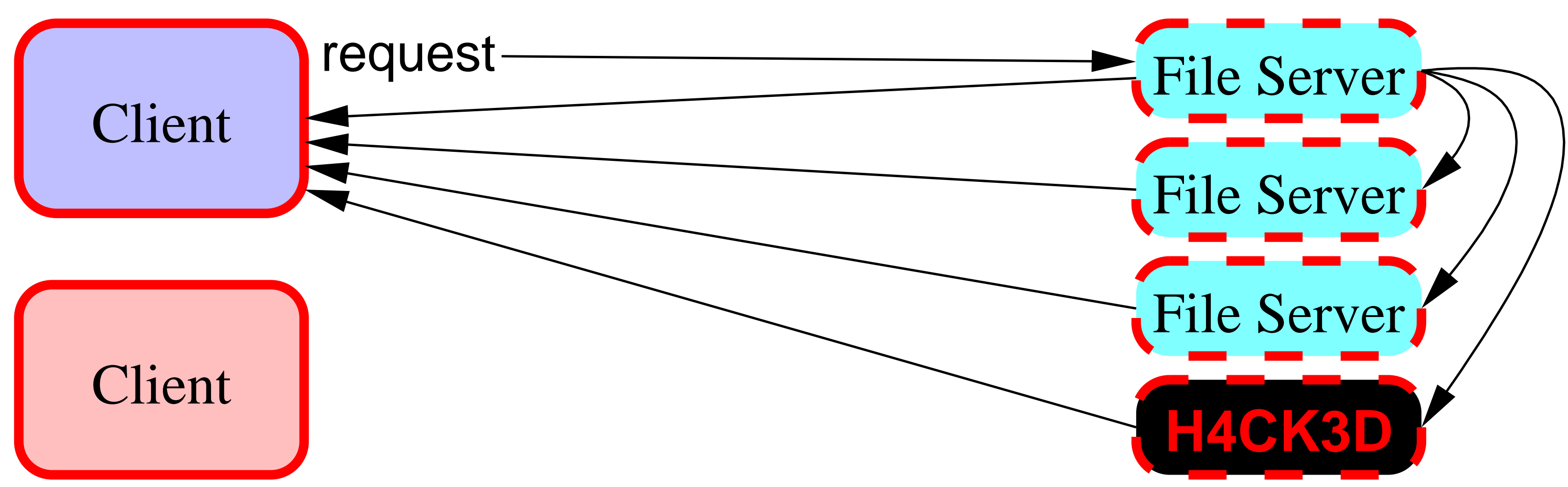
- **Honest server can & must reject bad client RPCs**
- **Bad server might collude with bad users**
  - Bad users can write some number of user & group i-handles
  - But "consistency" meaningless for bad-user-writable files (Technically already have permission to modify files between every pair of fetches by legitimate users)
  - And bad server alone can already raise "bad server" alert
- **What can server & clients do to files they can't write?**
  - Consider subset of operations on files bad users can't write
  - These operations will still be fork consistent

Scalability to multiple clients



- **Benchmark: unpack phase of emacs build**

BFS model



- **Replicate server 4 times**
  - Client sends request to replicas
  - 3 replicas must agree on order of the operation
  - 3 replicas must decide the operation will actually execute
  - Client waits for 2 such replicas to return identical responses
  - Okay if one replica compromised and/or one replica slow

SSL Convenience vs. Security

- **How convenient is a Verisign certificate?**
  - Need $300 + cooperation from NYU administrators
  - Good for credit cards, but shuts out many other people
- **How trustworthy is a Verisign certificate?**
  - In mid-March 2001, VeriSign, Inc., advised Microsoft that on January 29 and 30, 2001, it issued two… [fraudulent] certificates…. The common name assigned to both certificates is "Microsoft Corporation."

    VeriSign has revoked the certificates…. However… it is not possible for any browser's CRL-checking mechanism to locate and use the VeriSign CRL.

    – Microsoft Security Bulletin MS01-017
- **Is this the right level of protection for your data?**

Concurrent version structures

- **Define collision-resistant hash V for version structs**
  - E.g., delete i-handle, sort u-n/u-n-h data, run through H
- **Version structures now reflect pending updates**
  $\{\mathbf{VRS}, u_i\text{-}h, u_1\text{-}n_1\ \ldots\ u_i\text{-}n_i\ \ldots, u_1\text{-}n_1\text{-}h_1\ u_i\text{-}n_i\text{-}\perp\ \ldots\}_{K_{u_i}^{-1}}$
  - In addition to u-n pairs, v.s. has a u-n-h triple for each PVL entry
  - $u, n =$ user,version of a pending update
  - h is V of a version structure, or reserved "self" value $\perp$ (u's nth version structure always contains u-n-$\perp$)
  - Bump user + group #s, fold pending group ops into new i-handles!
- **View PVL as containing future version structures**
  - Each entry is of the form $\langle$update cert, $\ell\rangle$
  - ℓ is still unsigned version structure with i-handle $= \perp$
  - Clients compute each u-n-h triple with $V(\ell)$

Ordering concurrent version structures

Definition. We say $x \leq y$ iff:

1. For all users u, $x[u] \leq y[u]$ (i.e., $x \leq y$ by old def.), and
2. For each user-version-hash triple u-n-h in y, one of the following conditions must hold:
   (a) $x[u] < n$ (x happened before the pending operation that u-n-h represents), or
   (b) x also contains u-n-h (x happened after the pending operation and reflects the fact the operation was pending), or
   (c) x contains u-n-$\perp$ and $h = V(x)$ (x was the pending operation).

Signature speed

	Rabin	Esign	
	1,024 bits	2,048 bits	6,000 bits
Sign	3,656µs	169µs	695µs
Verify	27µs	120µs	575µs

- **Major cost of protocol is signatures**
  - One synchronous, one async signature per fetch/modify
  - But can amortize over many concurrent operations
- **Using Esign algorithm helps a lot**
- **Technology is on our side**
  - Digital signatures are getting faster & more secure
  - Speed of light is not changing
  - So eventually RTT will dominate public key crypto