Two ways to improve system security

	Trustworthy	**Untrustworthy**
Trusted	OK	BAD
Untrusted	OK	OK

- **Make components more trustworthy**
  - Fix bugs, simplify implementations, certify software, …
  - Sometimes makes it harder to innovate
- **Make components less trusted**
  - There are many untrusted resources out there…
  - if you can tap them, it may also enable new functionality

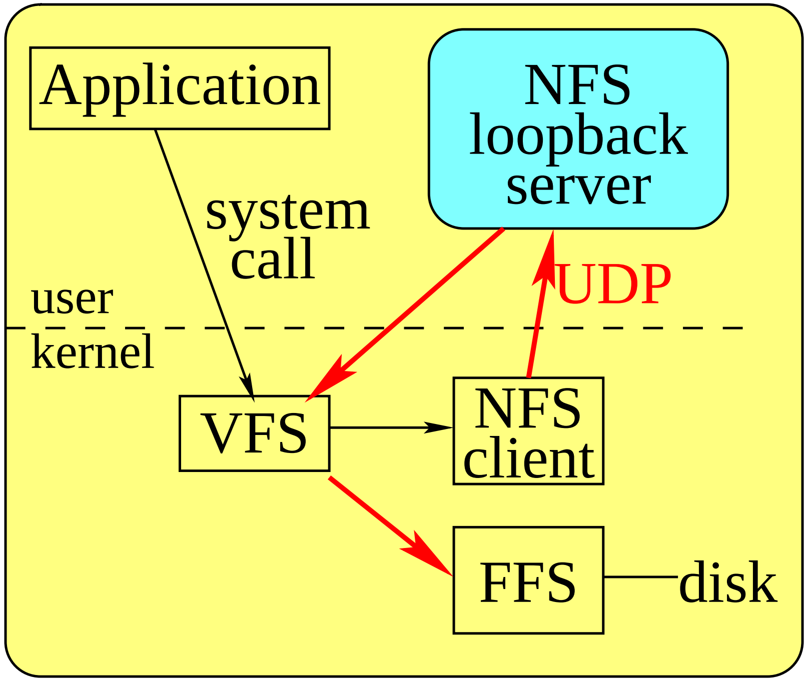
Medium-term plan

- **Next three lectures about untrusted components**
- **Today: Data security**
  - Secrecy of stored data on untrusted machines
  - Integrity of computation results on untrusted machines
  - Integrity of stored data on untrusted machines
- **Tuesday: Tamper-resistant computing**
  - Protecting against an attacker with physical control of a device
- **Next Thursday: Owner-resistant computing**
  - Viewing the legitimate owner of a computer as untrusted
  - (Is this even a good idea?)

Cryptographic Storage

- **Two models of cryptographic storage**
- **First model: Mitigate stolen computer / USB key**
  - Assume you will know when data is stolen
  - Once stolen, you no longer access compromised device
- **Second model: Outsource your data storage**
  - Store encrypted data on a server
  - Attacker may see multiple versions of data
  - Attacker may see access patterns
  - Very hard even to define security in this setting

CFS [Blaze]



- **Structured as *NFS loopback server***
  - Implement file system by speaking NFS over UDP
  - Encrypt contents of files as they are written
  - Must also encrypt file names, symbolic links, etc.

Example

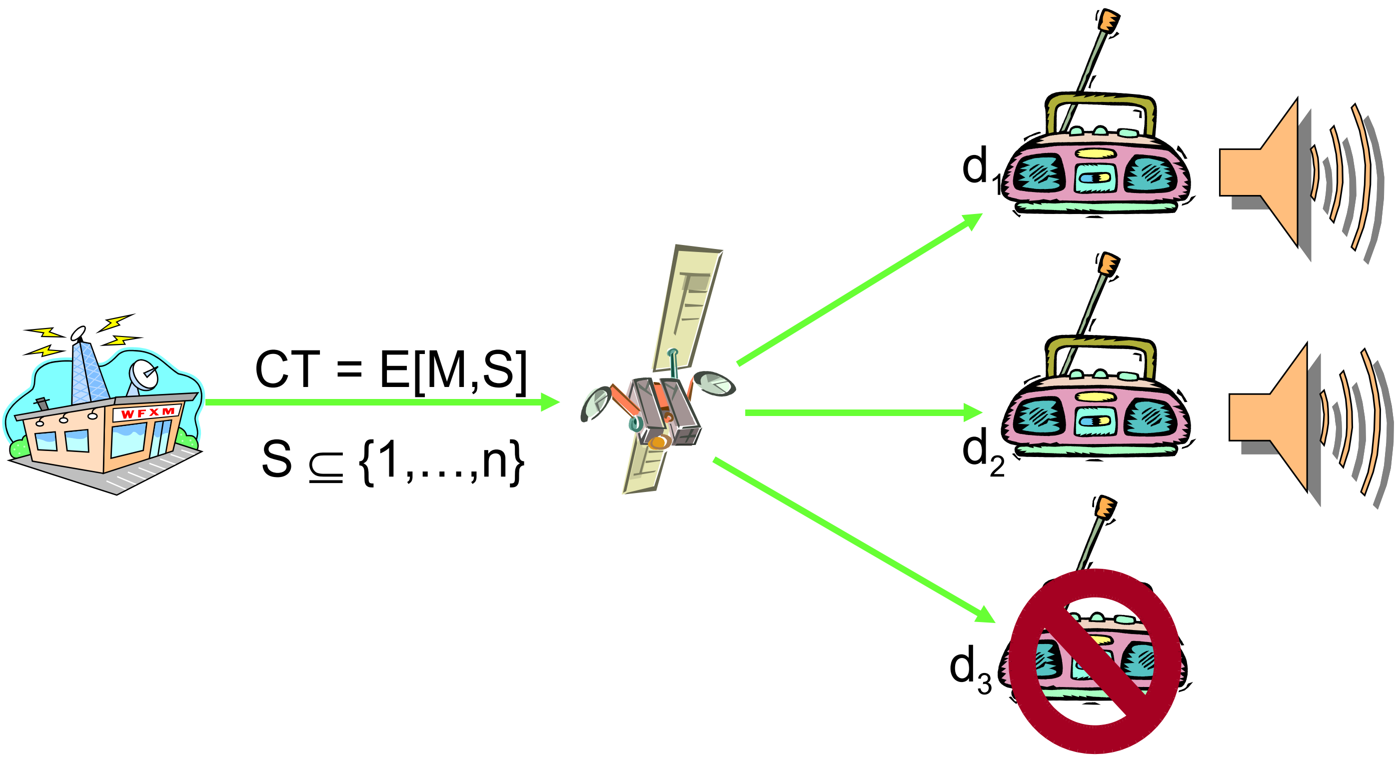
```
% cmkdir /usr/mab/secrets
Key:  (type password)
Again:  (type password)
% cattach /usr/mab/secrets matt
Key:  (type password)
% echo murder > /crypt/matt/crimes
% ls -l /usr/mab/secrets
-rw-rw-r-- 1 mab 15 Apr 1 15:57 8b06e85b87091124
% cat -v /usr/mab/secrets/8b06e85b87091124
M-Z,k^]^B^VM-VM-6A~uM-LM-_M-DM-^[
% detach matt
%
```

Initialization vectors

- **Recall encryption must be randomized**
  - E.g., if you copy a file, copy's ciphertext must look different
- **CFS solution: Use separate file for IV**
  - Makes operations like link, rename not atomic
  - On some benchmarks, cannot remove empty directories
- **Other solution: Store at beginning of file**
  - Reserve first 512 bytes for IV & other metadata
  - Performance impact is not too bad
  - Still need file for directory's IV

Sharing encrypted files

- **Must encrypt each file so only authorized readers can decrypt it**
- **Technique known as *broadcast encryption***
  - E.g., use for radio broadcast to paying subscribers



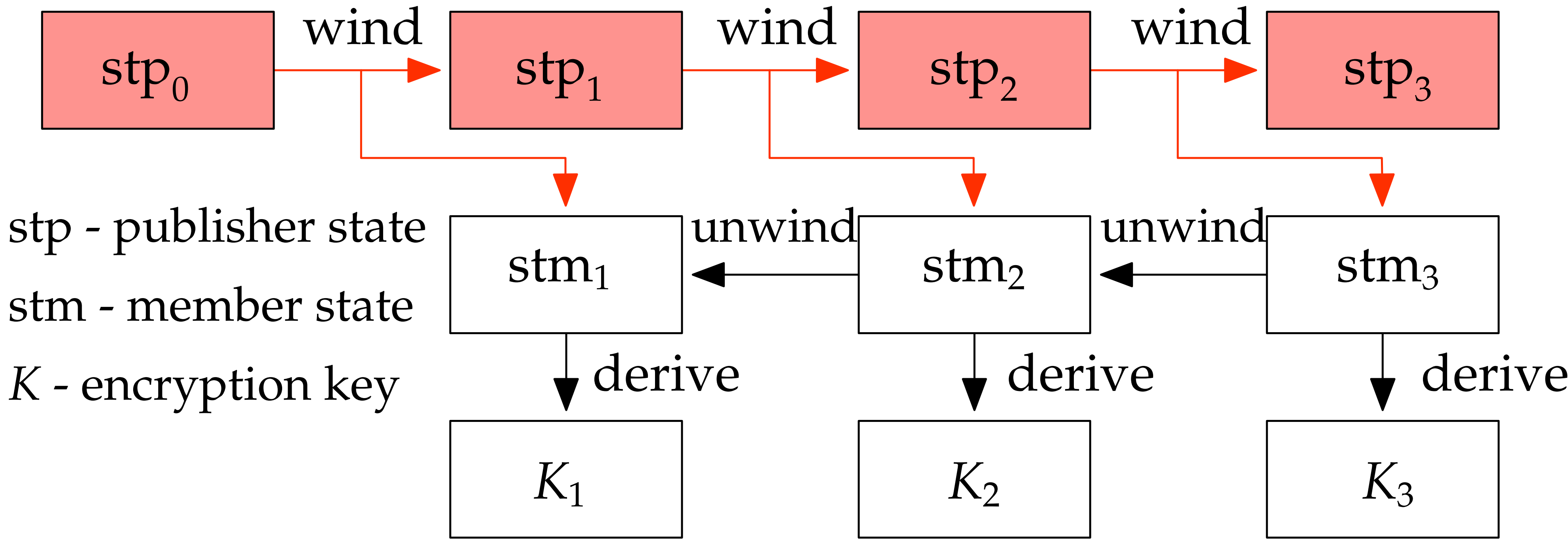
Broadcast encryption solutions

- **Small private key, large ciphertext**
  - Separately encrypt message for each recipient
  - E.g., limits number of people who can read file
- **Large private key, small ciphertext**
  - Use separate unique key for each set of recipients
- **Cryptographic techniques [Boneh, Gentry, Waters]**
  - Can make ciphertexts and private keys constant size
  - Just have to know for whom file encrypted to decrypt

Revocation

- **Want to revoke someone's read access from files**
- **One approach: re-encrypt all files w. new key immediately**
  - Potentially very expensive when kicking someone out of a widely-used group
  - Person may already have stored unencrypted copies of file anyway
- ***Lazy revocation*: encrypt all new content w. new key**
  - Ensures person can only read content from before revocation
- **Q: How to manage keys?**
  - People will need to read content encrypted w. old keys

Key regression [Fu et al.]



- **Switch from K_i to K_{i+1} when key revoked**
- **Give users state** stm$_{i+1}$
  - Can derive key K_i from state stm$_i$
  - Can also *unwind* stm$_i$ to any previous state
- **Only publisher can compute next member state**

Old state

- **What about data from before a user joins?**
- **In some cases, must prevent from reading**
  - E.g., Members of the Ph.D. admissions committee
  - Must read sensitive recommendation letters
  - Should not be able to read letters submitted about you
- **How to fix?**

Old state

- **What about data from before a user joins?**
- **In some cases, must prevent from reading**
  - E.g., Members of the Ph.D. admissions committee
  - Must read sensitive recommendation letters
  - Should not be able to read letters submitted about you
- **Might run two instances of key regression**
  - One "forwards" to current key
  - One "backwards" to when you joined
  - Derive real encryption key from forwards & backwards keys
- **Note: very bad if you have colluding users**

Byzantine Fault Tolerant Replication

Miguel Castro and Barbara Liskov

BFT replication

- **Goal: improve integrity of computation**
- **Idea: replicate server**
  - Attacker may be able to compromise one server
  - But compromising more than a fraction may be much harder
- **Structure server as a deterministic state machine**
  - If each correct replica executes the same operations, will return the same results
- **System must handle *Byzantine failures* of replicas**
  - Most systems expect fail-stop behavior (black smoke)
  - Byzantine failure means server can give you bad responses

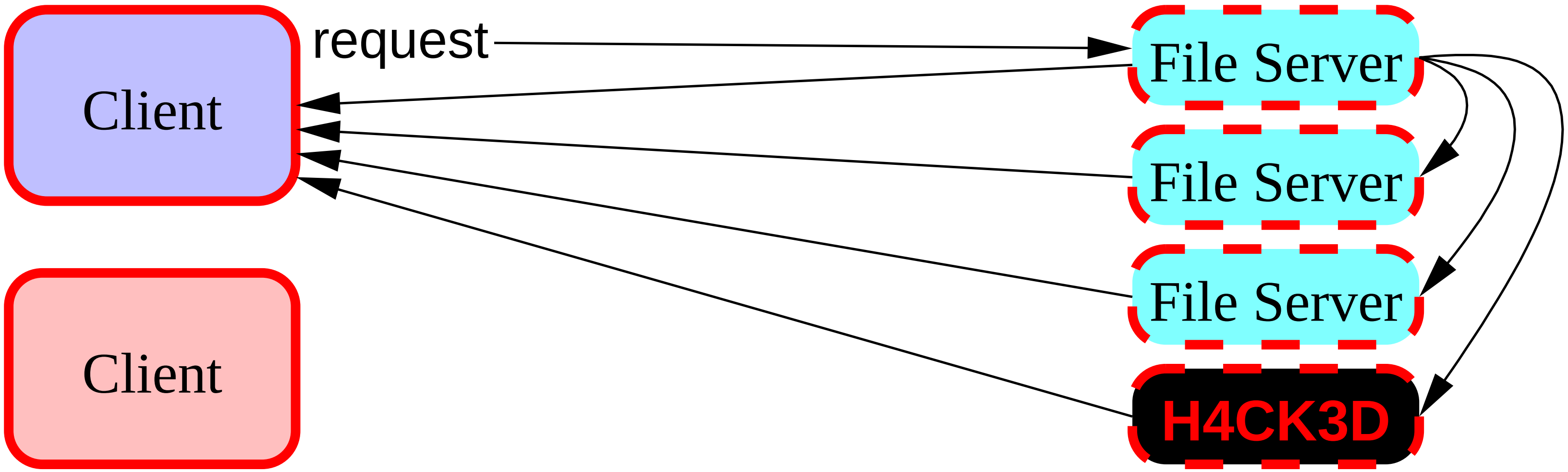
Straw-man BFT replication

- **Replicate server on three machines**
- **Assume at most one will be compromised**
- **For each operation:**
  - Broadcast request to all three replicas
  - If they differ in their replies, go with the majority
- **What's wrong here?**

BFT replication complications

- **Replicas must somehow agree on order of operations**
  - Otherwise, will get out of sync
- **Failed and slow replicas are indistinguishable**
  - Say you hear back from replicas 1 and 2 but not 3
  - 3 may have failed, so want to proceed
  - But what if 2 has actually failed, and 3 is just slow
  - If you proceed, honest replicas 1 and 3 will be out of sync
  - So at very lease replica 2 can cause divergent views

BFT overview



- **Replicate server $3f + 1$ times to tolerate f faults**
  - Client sends request to replicas
  - $2f + 1$ replicas must agree on order of the operation
  - $2f + 1$ replicas must decide the operation will actually execute
  - Client waits for $f + 1$ such replicas to return identical responses
  - Okay if f replicas compromised and/or f replicas slow

PBFT (simplified)

1. $c \to R$: $m = \{\textbf{REQUEST}, o, t, c\}_{K_c^{-1}}$
   - Client c broadcasts request o to set of all replicas R
   - Signs message, includes unique timestamp t
2. $p \to R$: $\{\textbf{PRE-PREPARE}, v, n, d = H(m)\}_{K_p^{-1}}$
   - Replicas proceed through sequence of *views*
   - In view number v, replica $v \bmod (3f+1)$ is primary
   - Primary picks sequence number n for m & broadcasts it
3. $r_i \to R$: $\{\textbf{PREPARE}, v, n, d, i\}_{K_{r_i}^{-1}}$
   - Each replica promises not to accept operation other than d for sequence number n in view v

PBFT (continued)

- **Say** $\mathit{prepared}(m, v, n, i)$ **when replica** i **has** $2f + 1$ **matching PREPARE messages (including its own)**
  - Means $\mathit{prepared}(m', v, n, j)$ w. $m \neq m'$ false for any honest r_j
- **But not safe to execute operation yet!**
  - Just because another m' won't execute doesn't mean m will
  - Might be view change if primary is faulty
- **Execute when** $\mathit{prepared}(m, v, n, i)$ **true for** $2f + 1$ **(meaning** $f + 1$ **non-faulty) replicas** r_i
  - Note: means any 2f+1 replicas will contain one honest replica that can prove no other m' executed at n in v
- **Say** $\mathit{committed}(m, v, n)$ **when okay to execute**
  - How does a replica r_i know $\mathit{committed}(m, v, n)$?

PBFT (continued)

4. $r_i \longrightarrow R$**:** $\{\textbf{COMMIT}, v, n, d, i\}_{K_{r_i}^{-1}}$

- r_i sends COMMIT message once *prepared*(m, v, n, i)
- Waits for $2f + 1$ matching COMMITs, including its own; once received, we say *committed-local*(m, v, n, i)
- *committed-local*(m, v, n, i) implies *committed*(m, v, n)

5. $r_i \longrightarrow c$**:** $\{\textbf{REPLY}, t, c, \text{result}, i\}_{K_{r_i}^{-1}}$

- Execute operation and reply once *committed*(m, v, n)
- Client c waits for $f + 1$ matching replies (meaning at least one is from honest replica)

View changes

- **Must change views if primary is bad**
  - Replicas may notice primary not responsive
  - $f + 1$ replicas suspecting primary should trigger view change
- $r_i \rightarrow R$**:** $\{\textbf{VIEW-CHANGE}, v+1, n, \mathcal{C}, \mathcal{P}, i\}_{K_{r_i}^{-1}}$
  - $\mathcal{P}$ is $2f + 1$ matching PREPAREs for all messages where $\mathit{prepared}(m, v, n, i)$
  - [Actually, $\mathcal{C}$ is checkpoint so don't need whole history in $\mathcal{P}$]
- $p' \rightarrow R$**:** $\{\textbf{NEW-VIEW}, v+1, V, O\}$
  - p' is new primary
  - V is set of $2f + 1$ view change messages
  - O is PRE-PREPAREs for messages in $\mathcal{P}$s

SFSRO

M. Frans Kaashoek, Kevin Fu, and David Mazières

Content distribution problem

- **People often distribute popular files from mirrors**
  - Have files been tampered with?

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

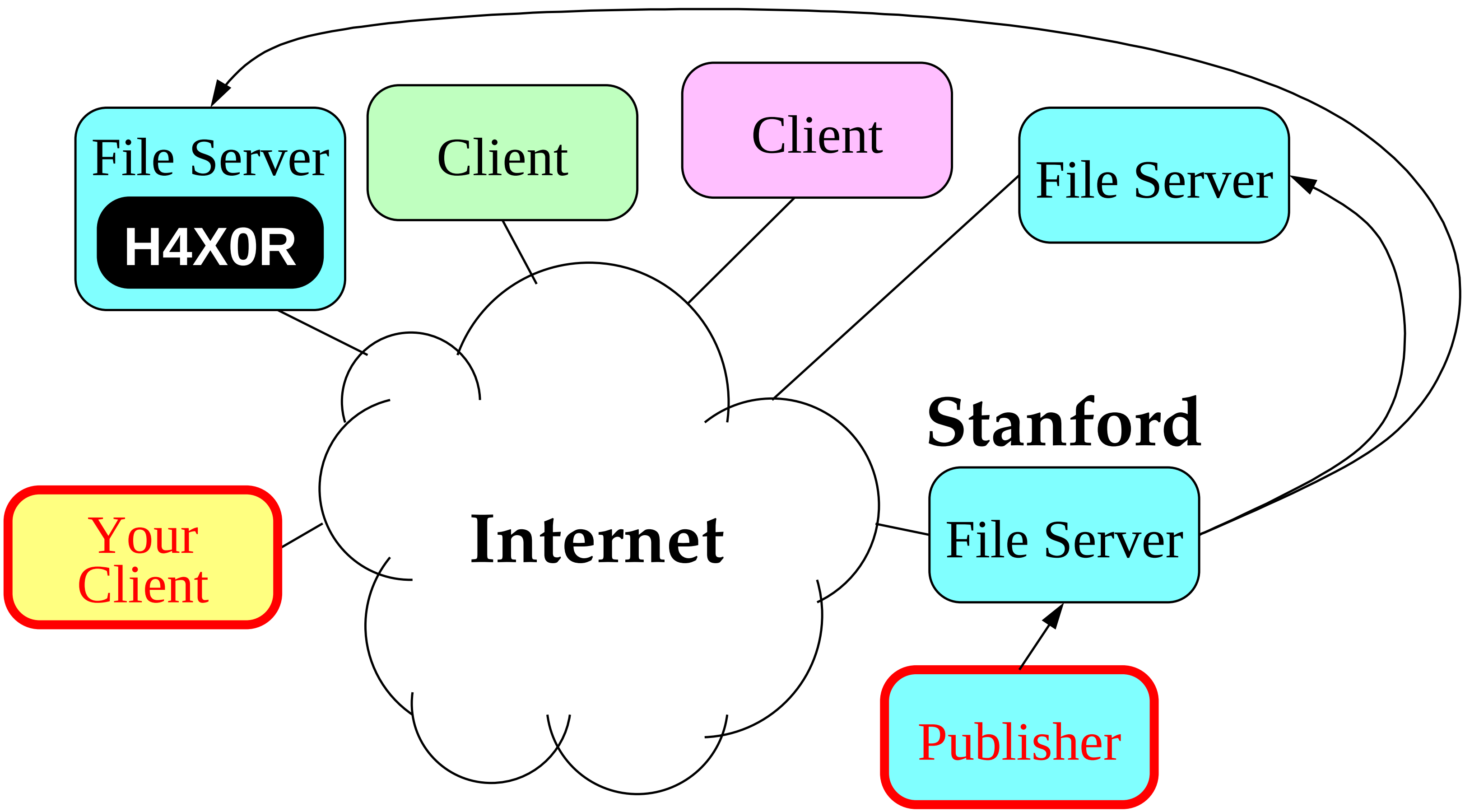
Signing individual files

- **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?**

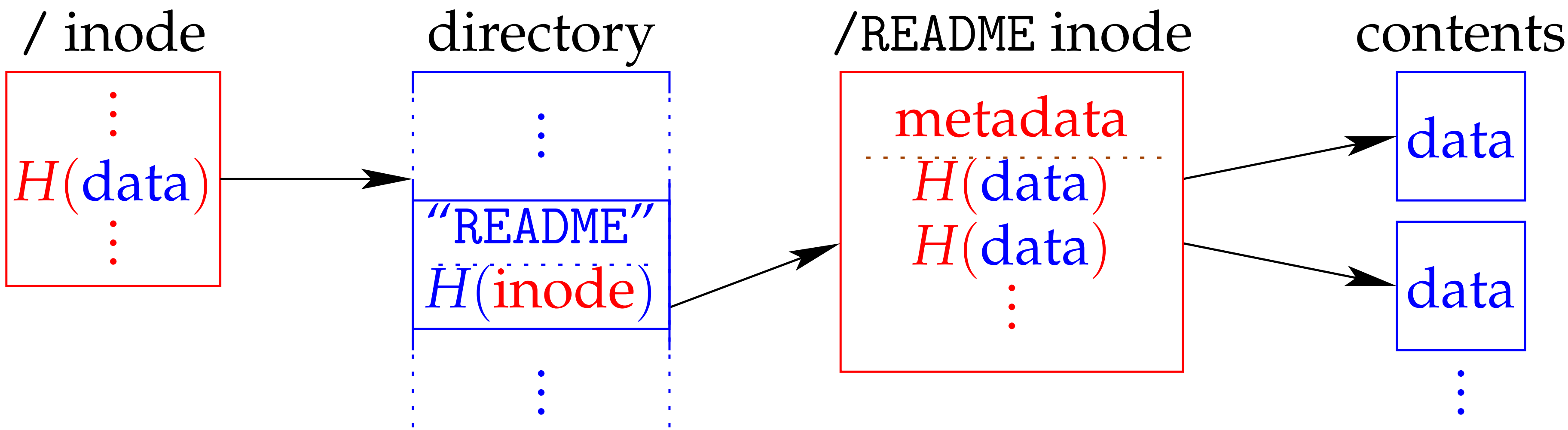
SFSRO: Signing whole file systems



- Give publisher a signature key (public key in path)
- 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

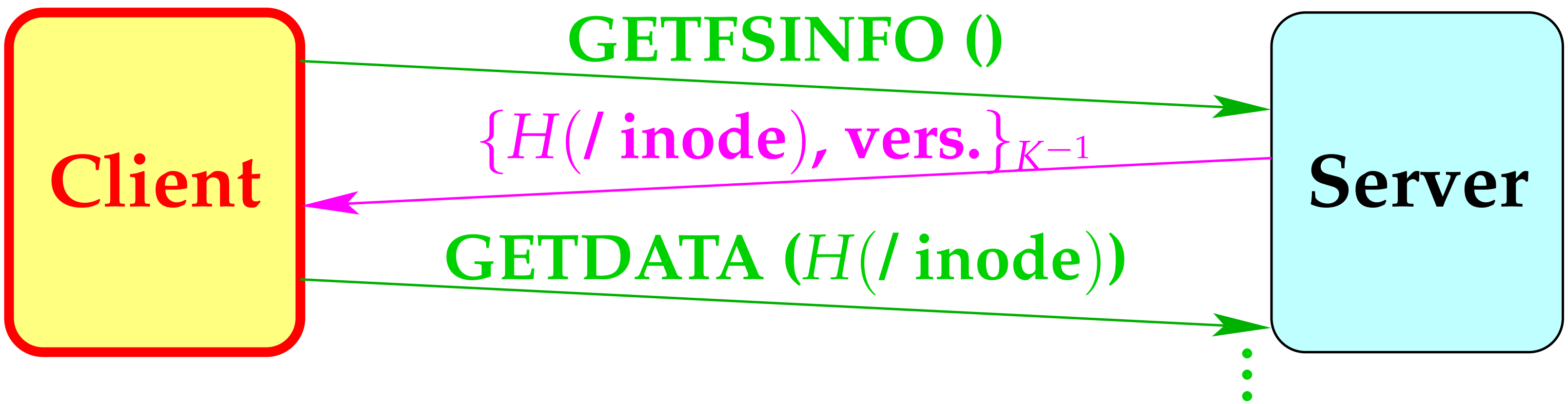
- **Can't just sign raw disk image (too big)**
  - Users may want to download and verify only a few files



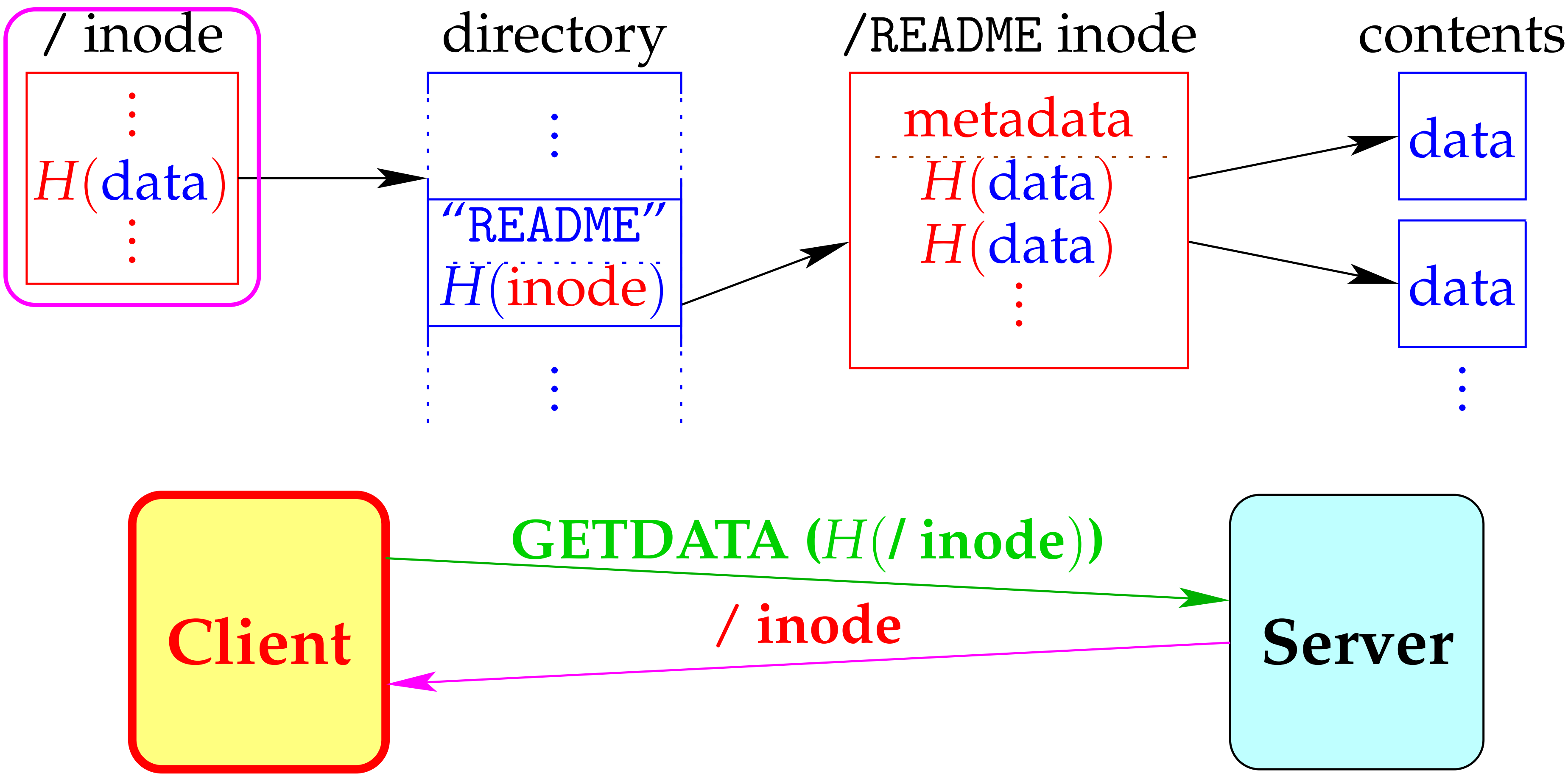
  - H is a collision-resistant hash function w. fixed-size output
- **Publisher signs hash of root inode**
- **Idea influenced many systems (CFS, Venti, . . .)**

SFSRO Protocol

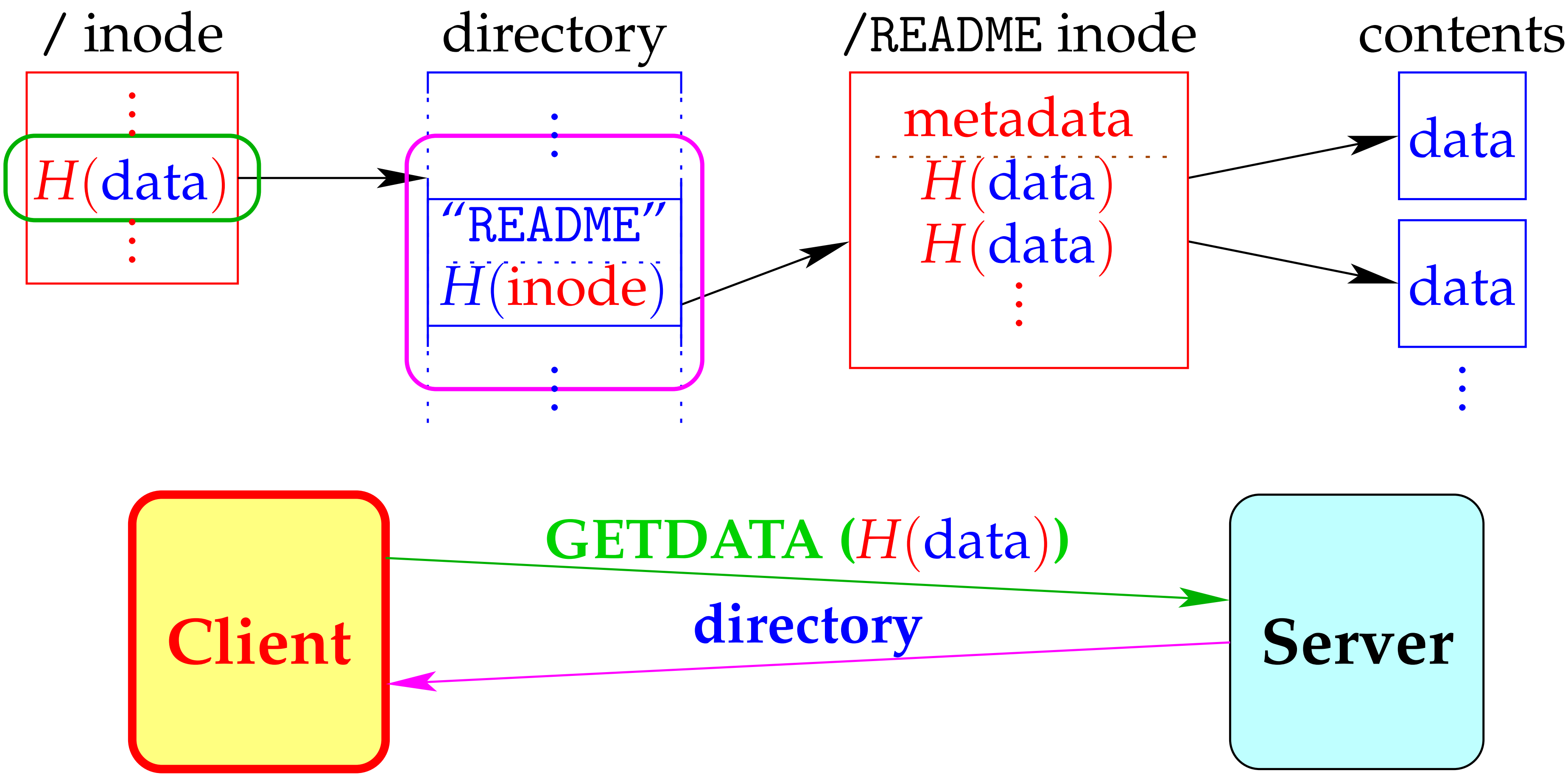
- **GETFSINFO () – Get signed hash of root directory**
- **GETDATA (*hash*) – Get block with *hash* value**
- **Example: To read file** `/README`
  - First get signed hash, then walk down tree



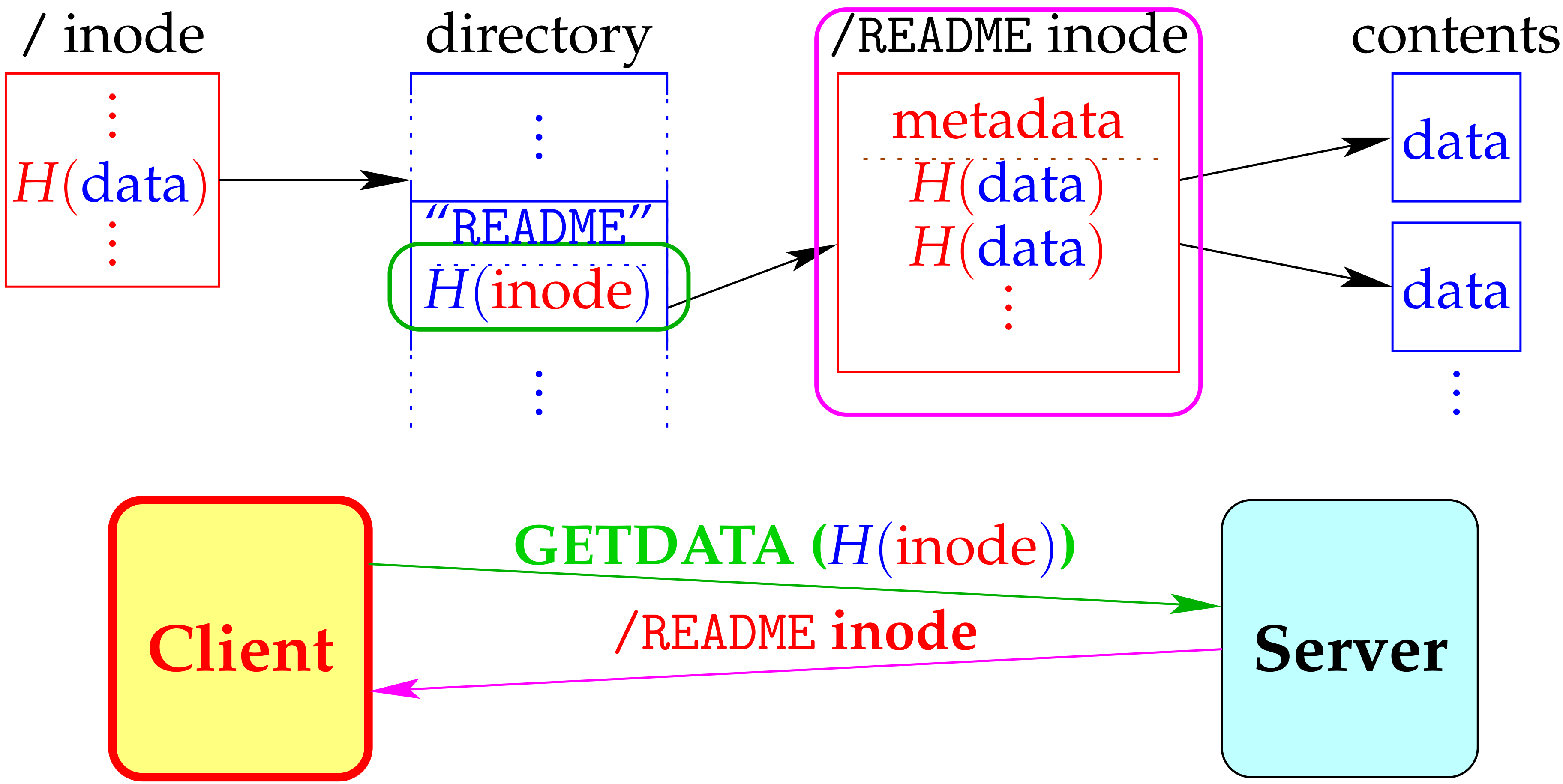
SFSRO Protocol



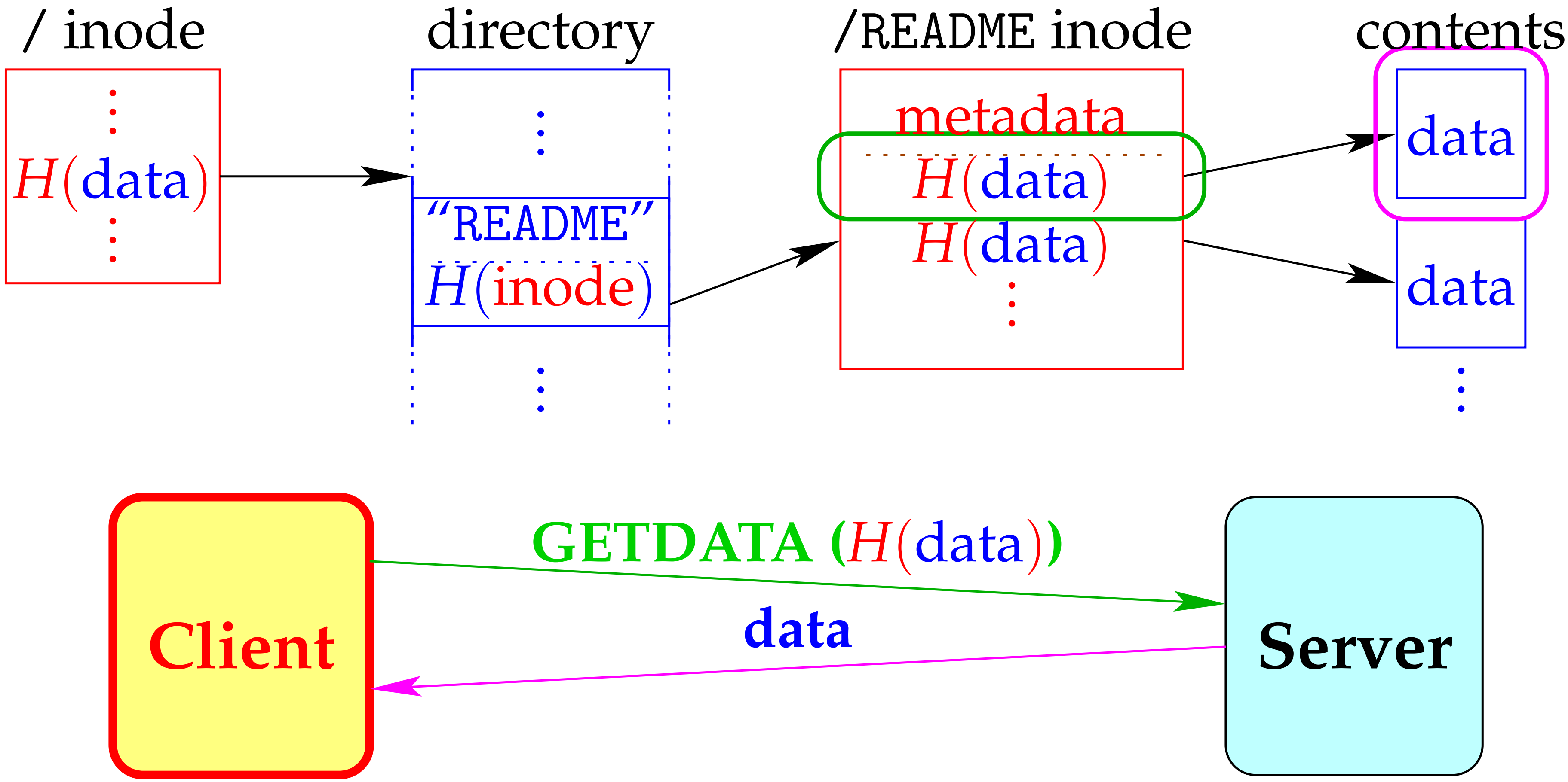
SFSRO Protocol



SFSRO Protocol



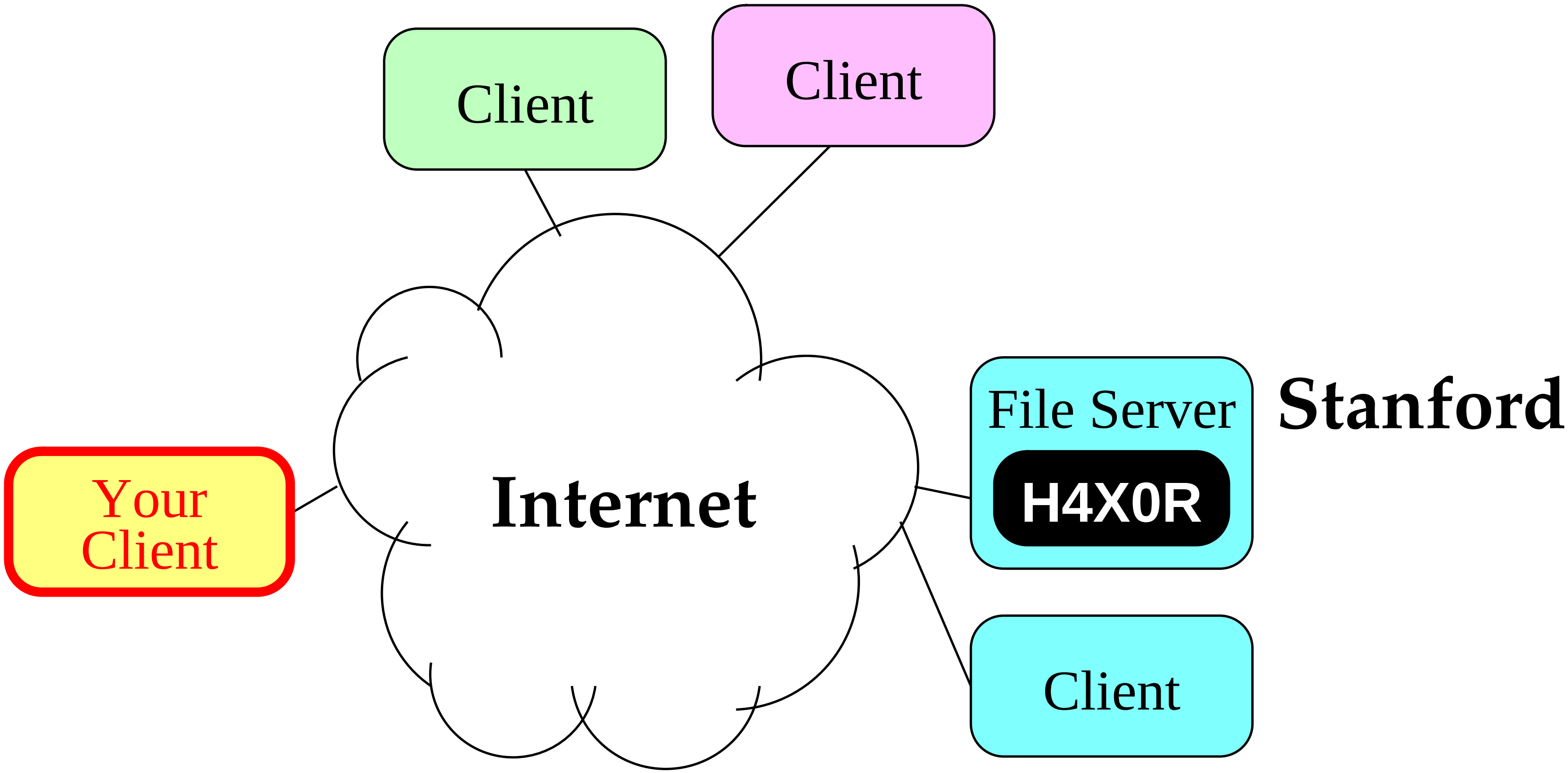
SFSRO Protocol



SUNDR

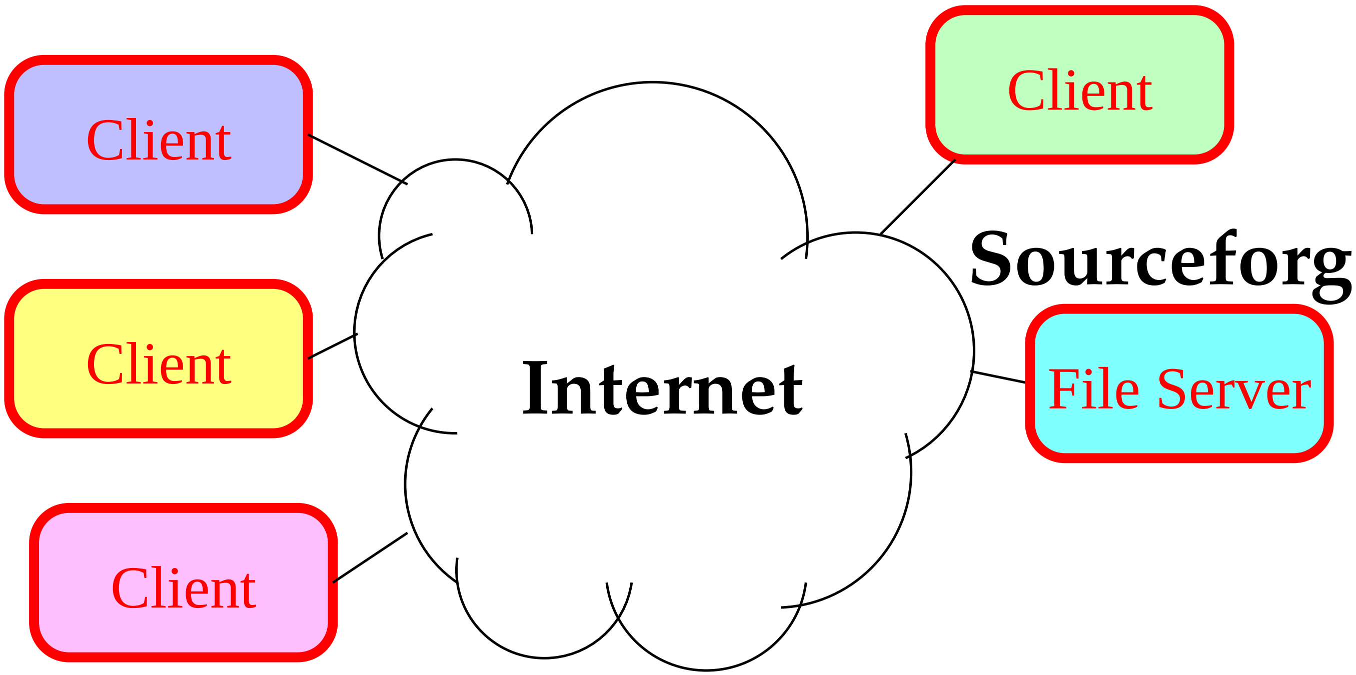
Jinyuan Li, Max Krohn, David Mazières, and Dennis Shasha

SUNDR: End-to-end FS integrity



- **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

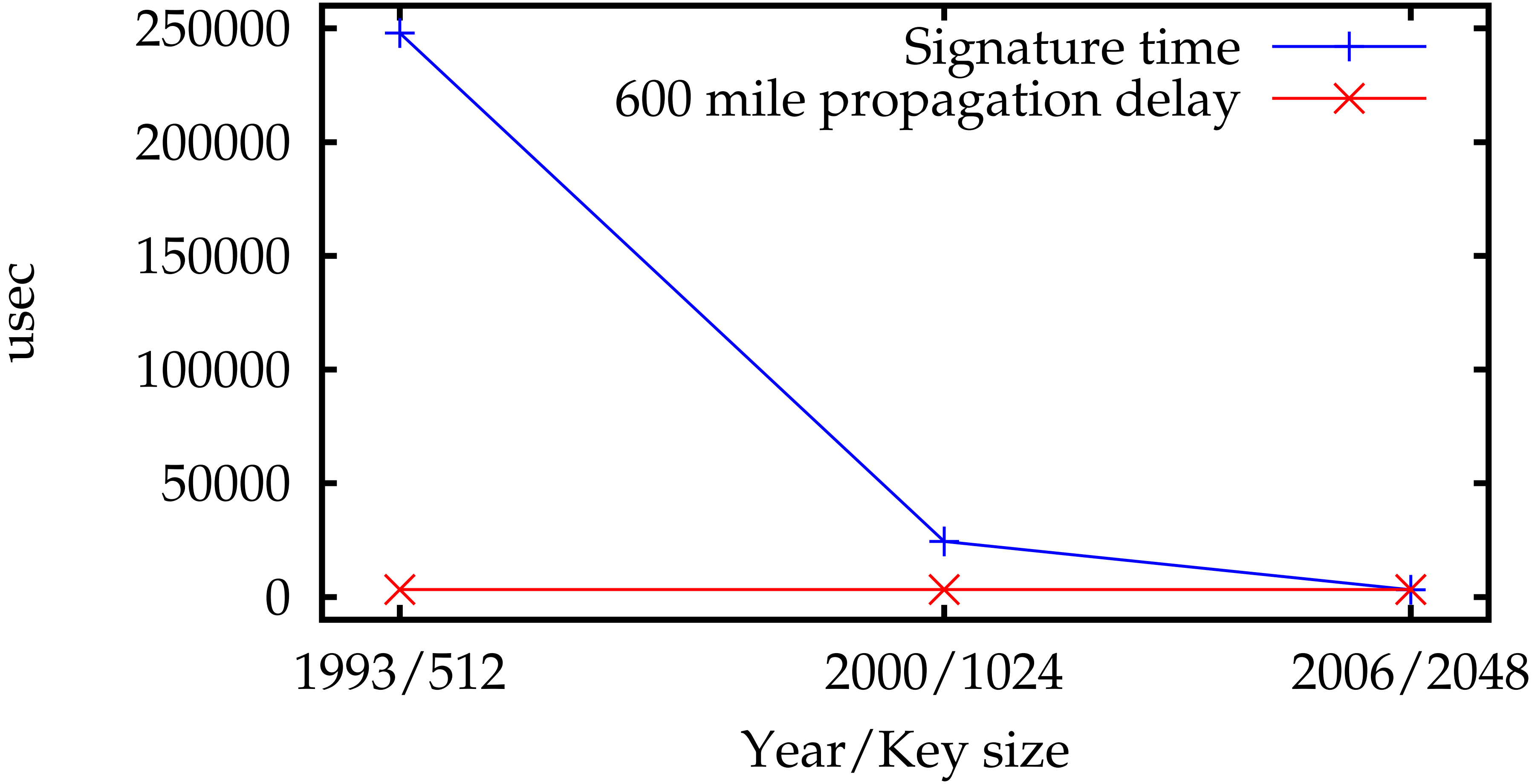


- E.g., Sourceforge hosting source repositories
- Attractive target of attack

A worrisome trend

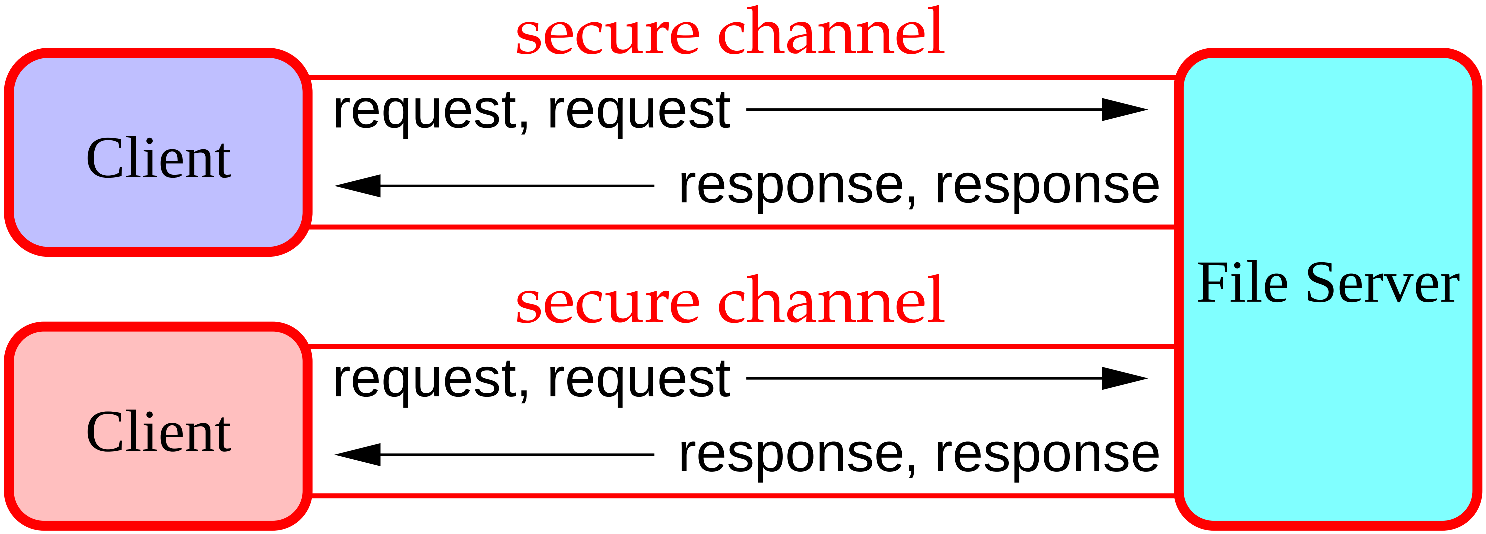
- **5/17/01: Apache development servers compromised**
  - Password captured by trojaned ssh binary at sourceforge
  - The integrity of all source code repositories is being individually verified by developers… - Apache press release
- **11/20/03: Debian administrators discover "root kit"**
  - at the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

Good News: Digital Signature Cost



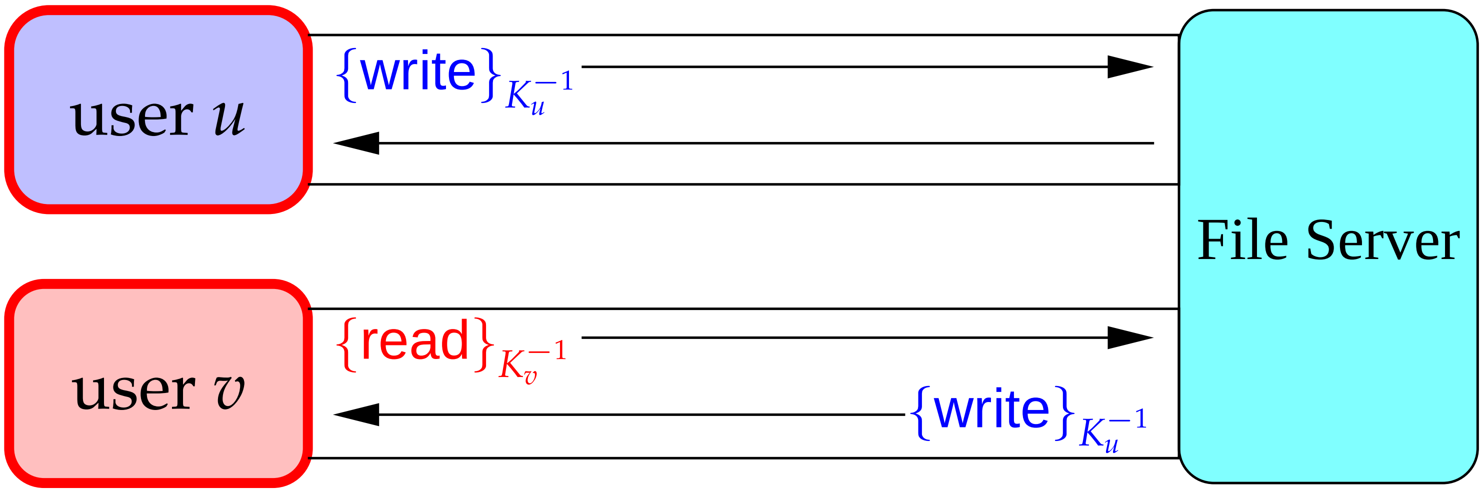
- **Signing every network request soon practical**

Traditional file system model



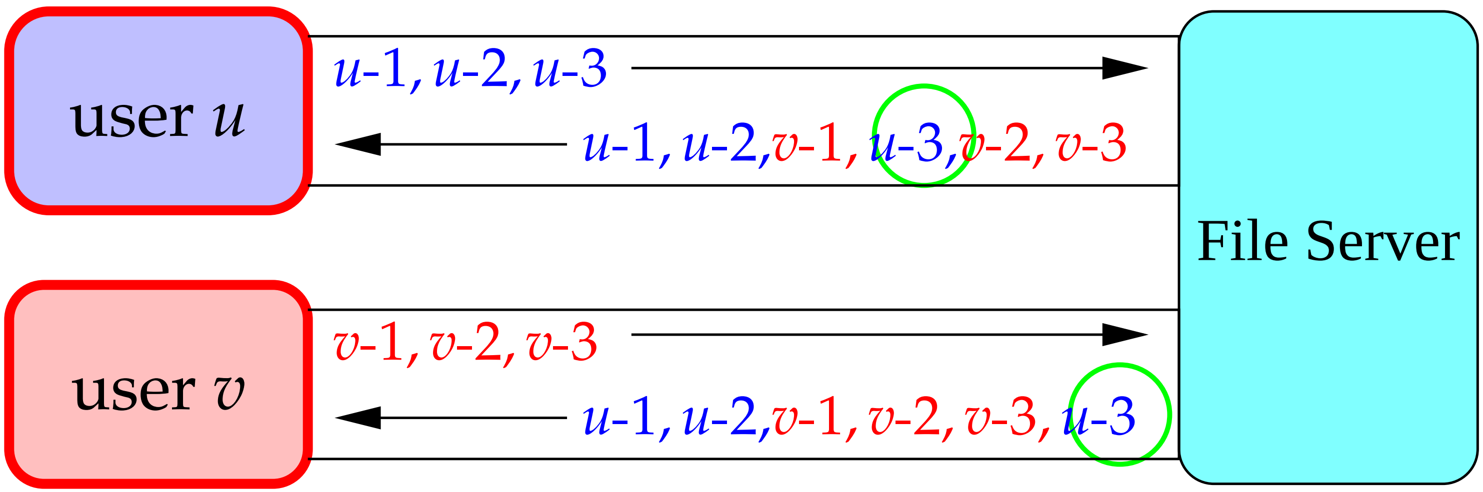
- **Clients & servers communicate over secure channels**
  - Network attackers can't tamper with requests
- **Server can't prove what requests it received**
  - Trust server to execute requests properly
  - Trust server to return correct responses

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

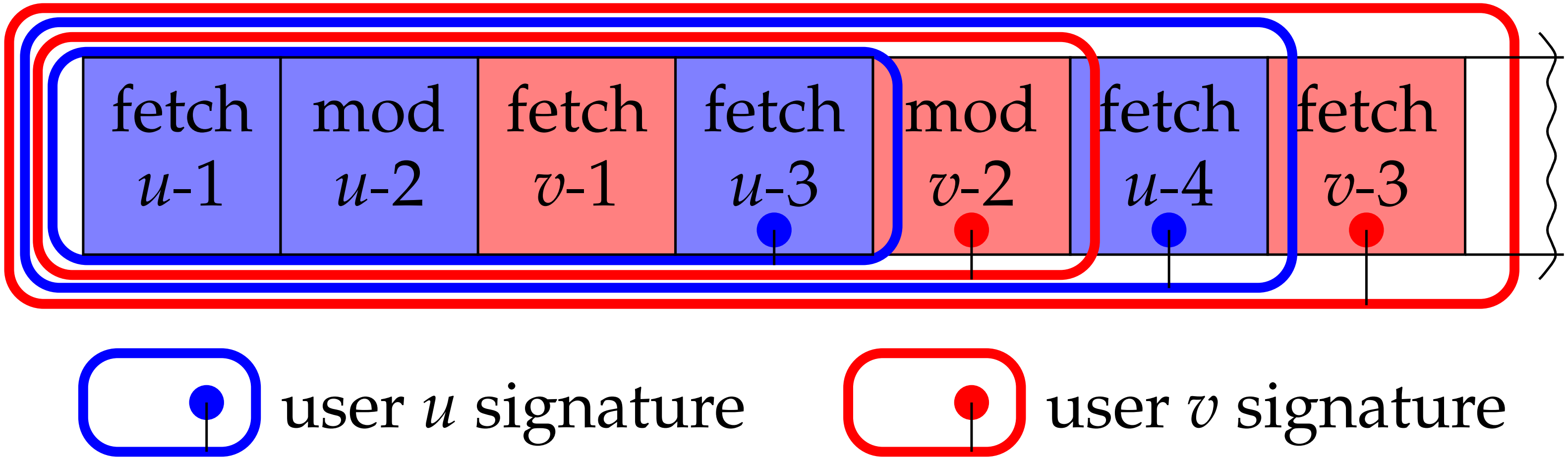
A Fetch-Modify interface

- **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: 2 or 3 progressively more realistic realizations**
  - Signed logs (enormous bandwidth & FS-wide lock)
  - Serialized SUNDR (FS-wide lock)
  - SUNDR (if we have time)

Solution 1: Signed logs



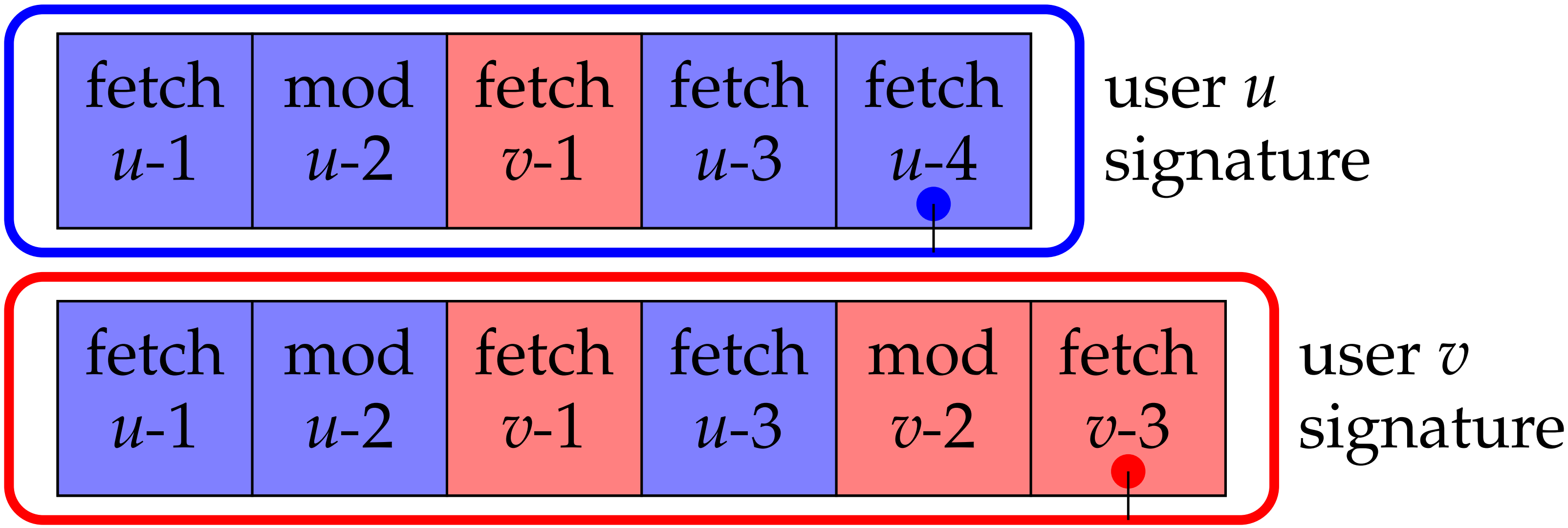
- **Detect reordering by signing entire FS history:**
- **PREPARE RPC – lock file system, download log**
  - Client checks signatures on log entries
  - Client checks that its previous operation is still in log
- **Client plays log to reconstruct FS state**
- **Client appends new operation, signs new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

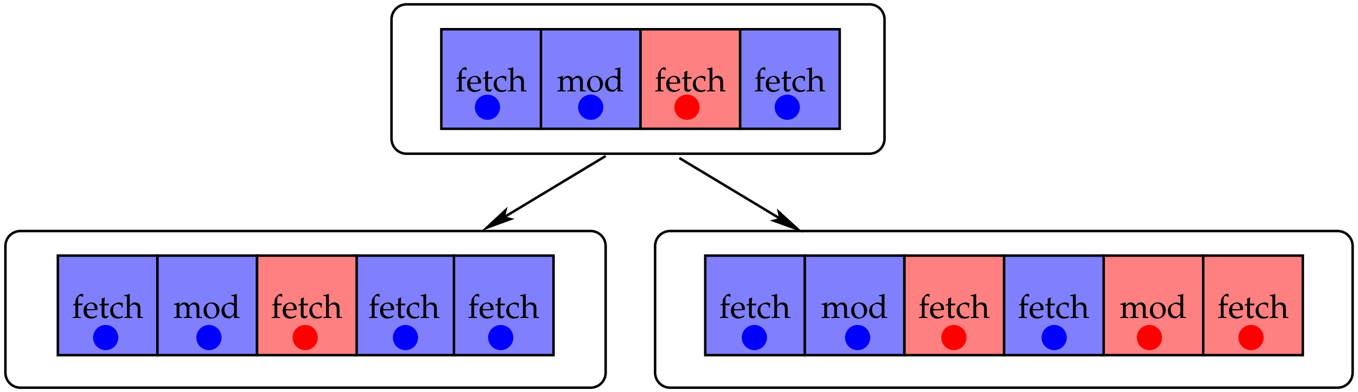
What can a malicious server do?

- **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

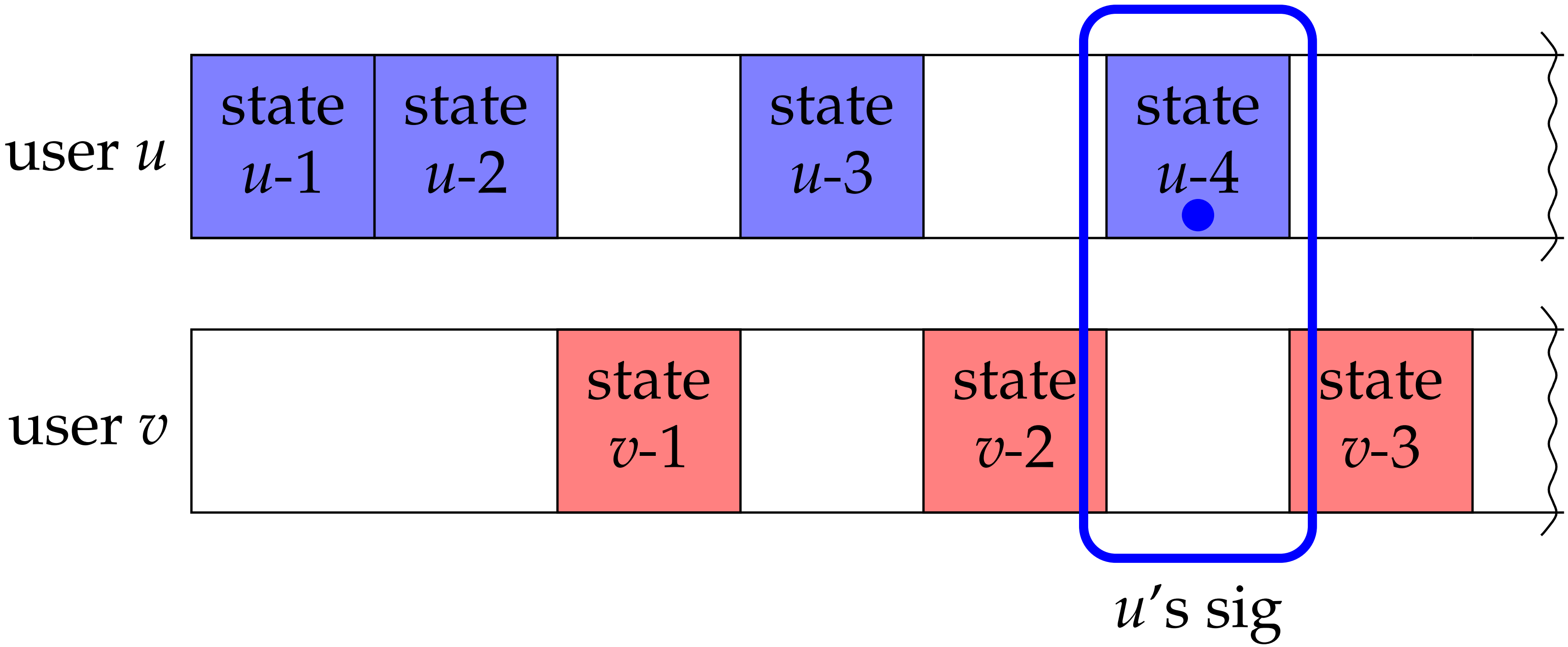
Implications of fork consistency

- **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?

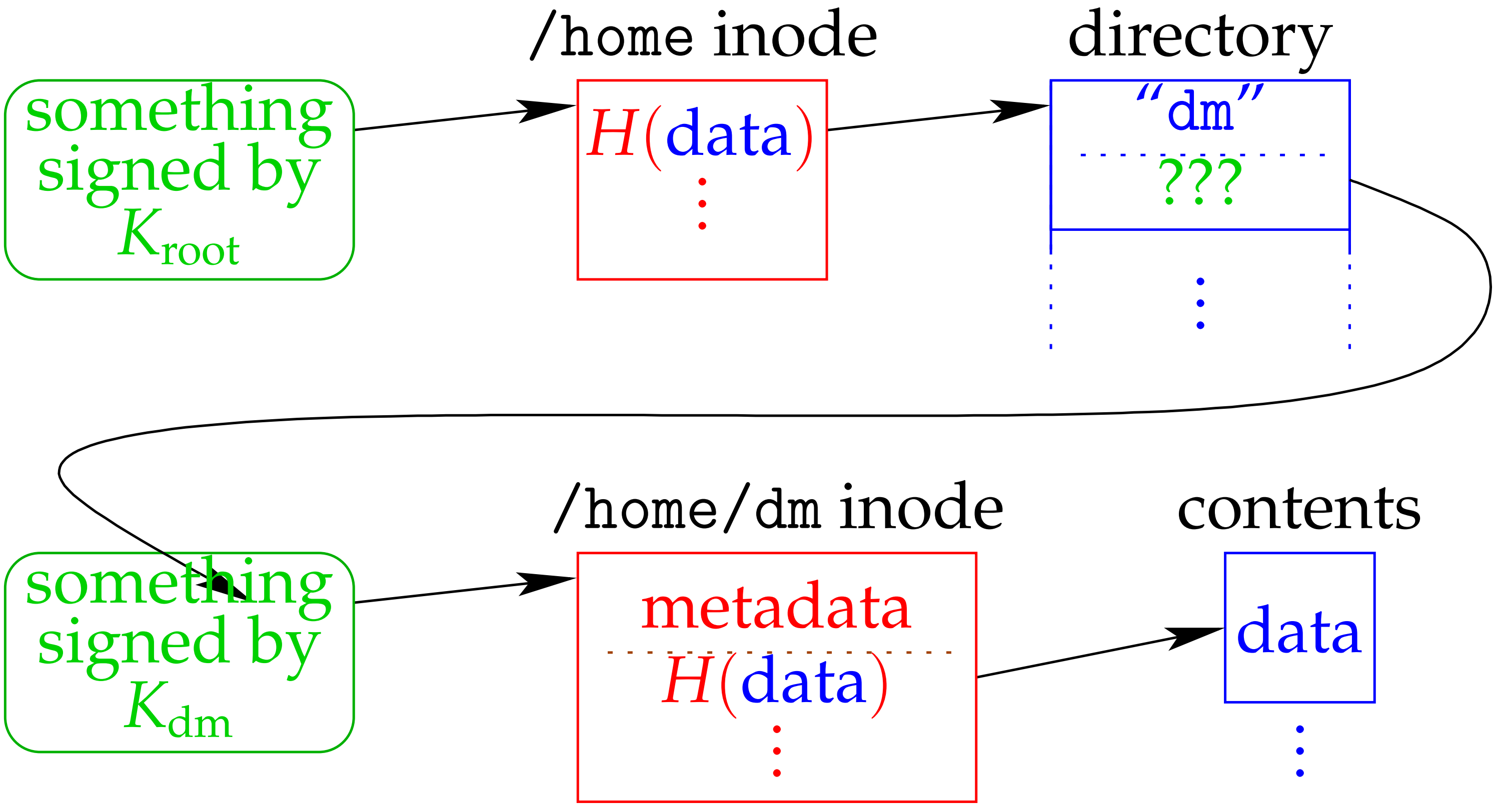
A plan for signing snapshots



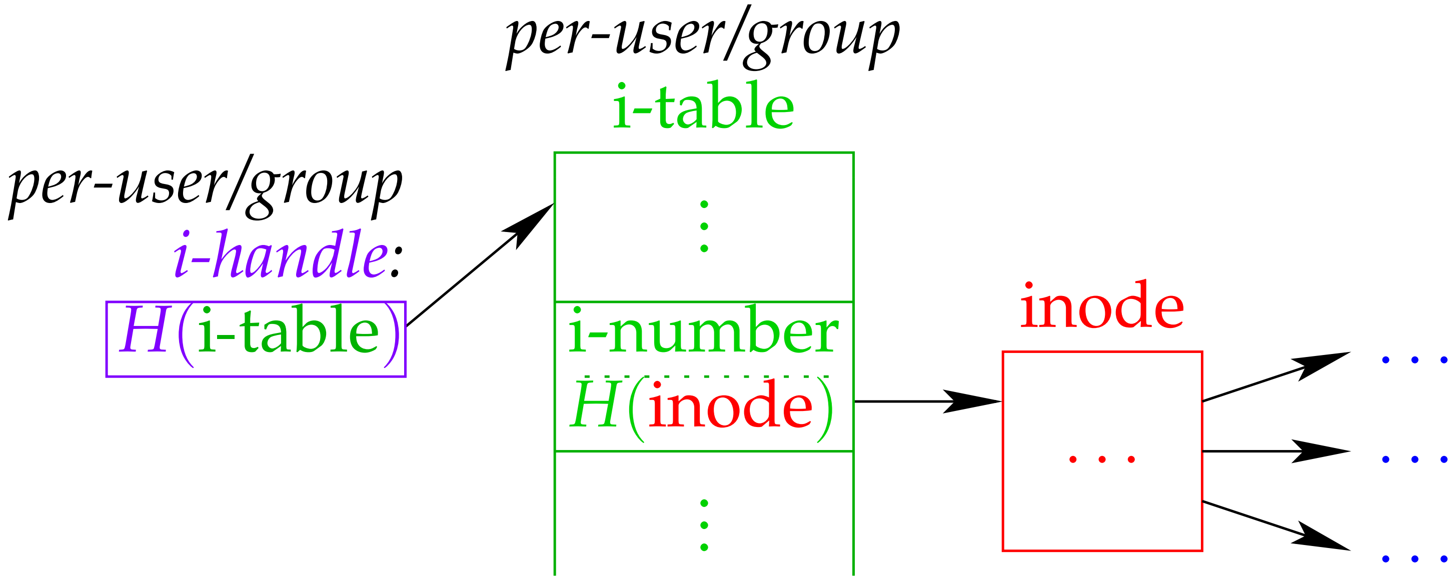
- **Somehow represent snapshots of each user's files in a way that they can be combined…**
- **Somehow prevent re-ordering of users' snapshots…**

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

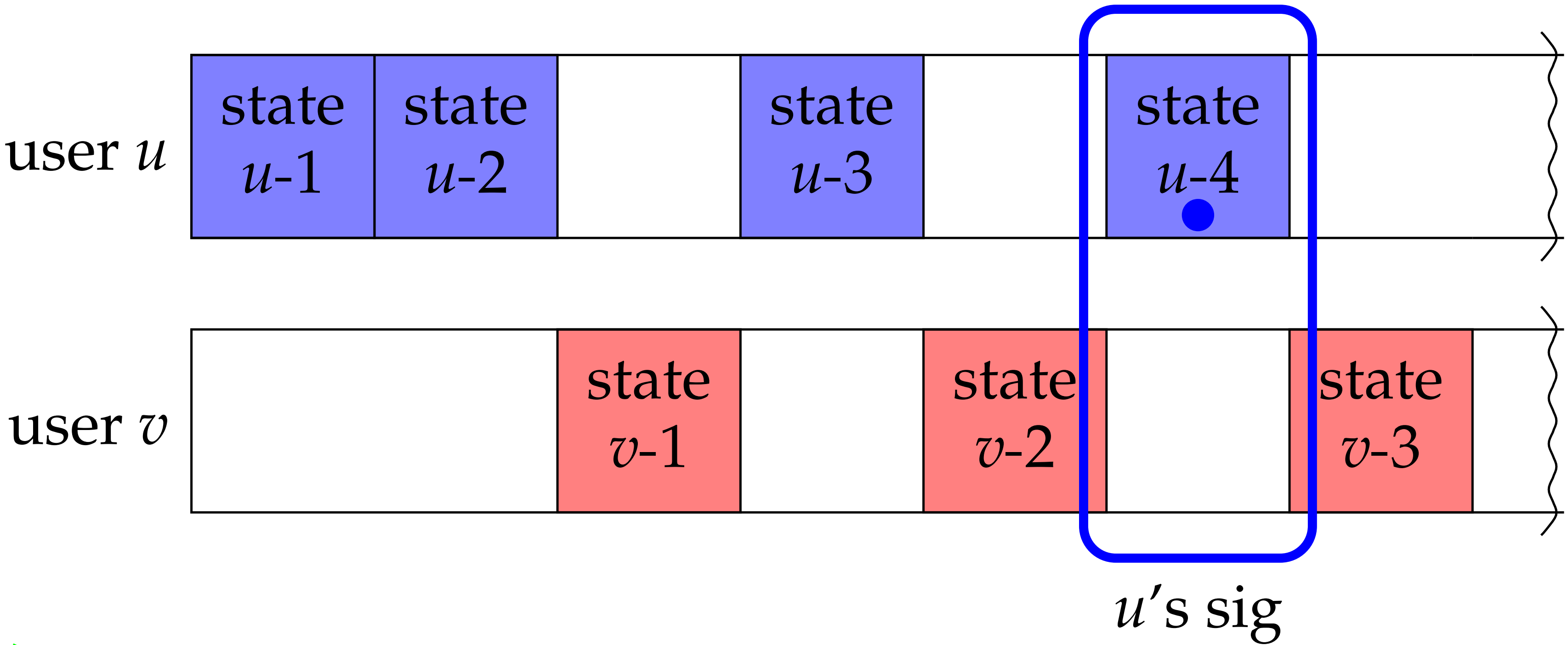


Per-user or -group i-numbers



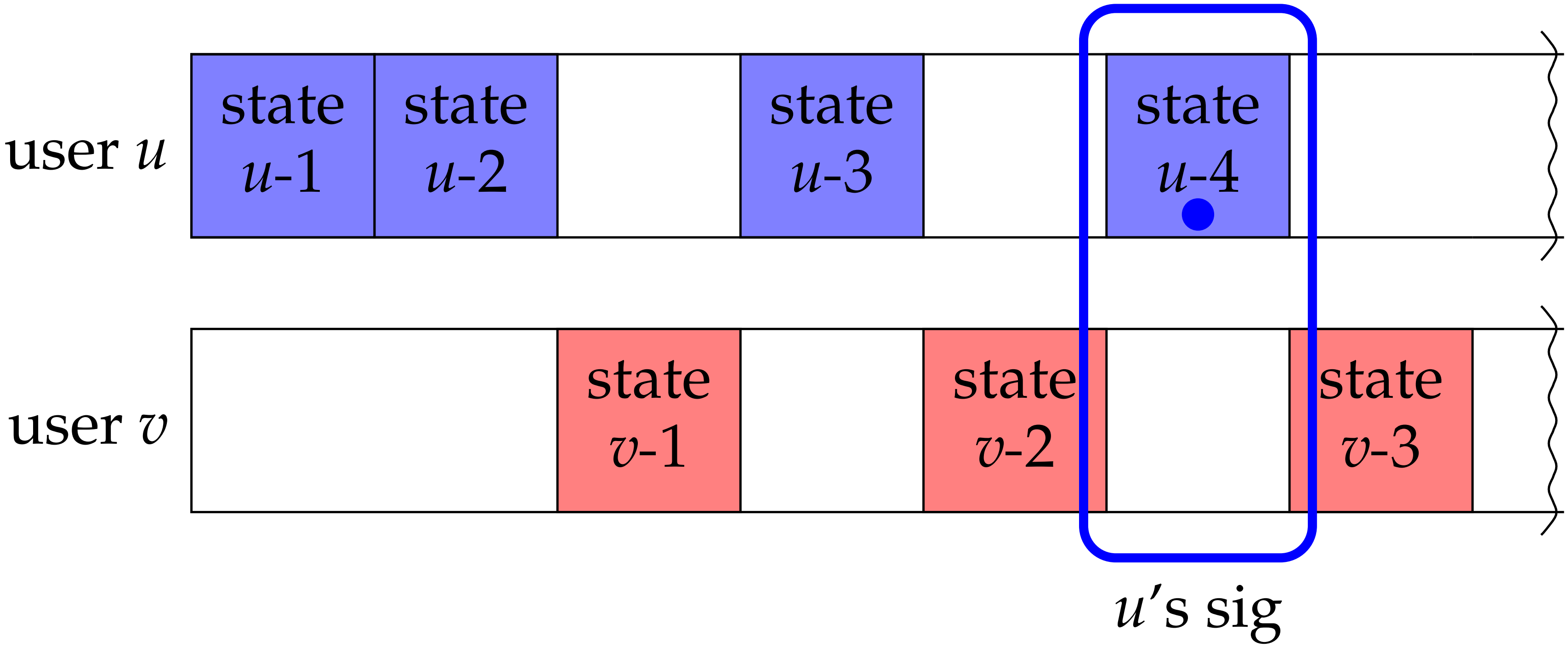
- **Add a level of indirection to SFSRO data structures**
- **SUNDR directory entry:** file name / ⟨user/group, i-number⟩
- **Per-user/group *i-tables* map** i-number $\rightarrow H(\text{inode})$
- **Hash each** i-table **to a short *i-handle* users can sign**

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 w. version vectors



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

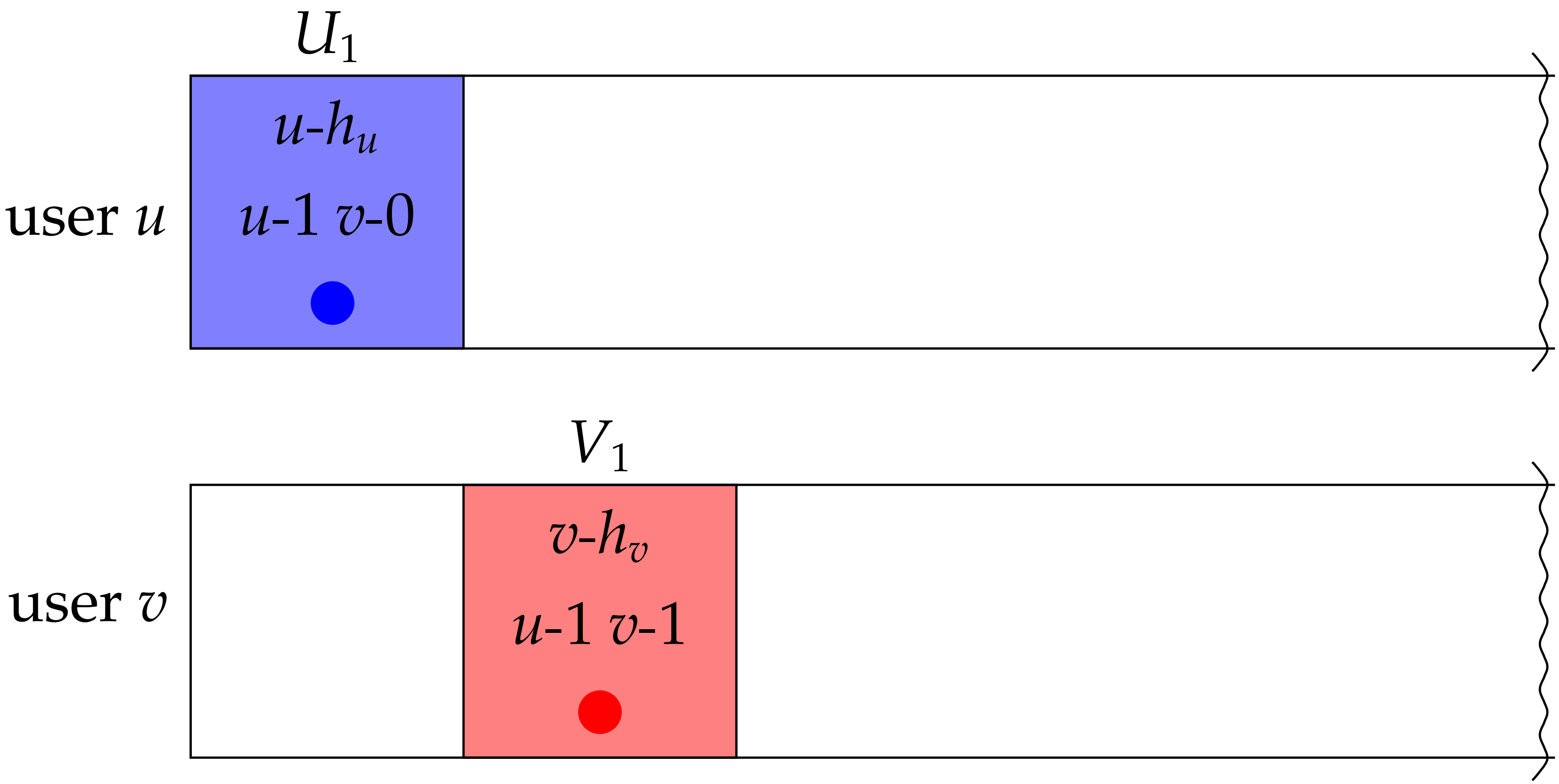
$$\textit{version structure:}\ \{\ \underbrace{u\text{-}h_u}_{\text{i-handle}}\ ,\ \underbrace{u\text{-}4\ v\text{-}2}_{\text{version vector}}\ \}_{K_u^{-1}}$$

- **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

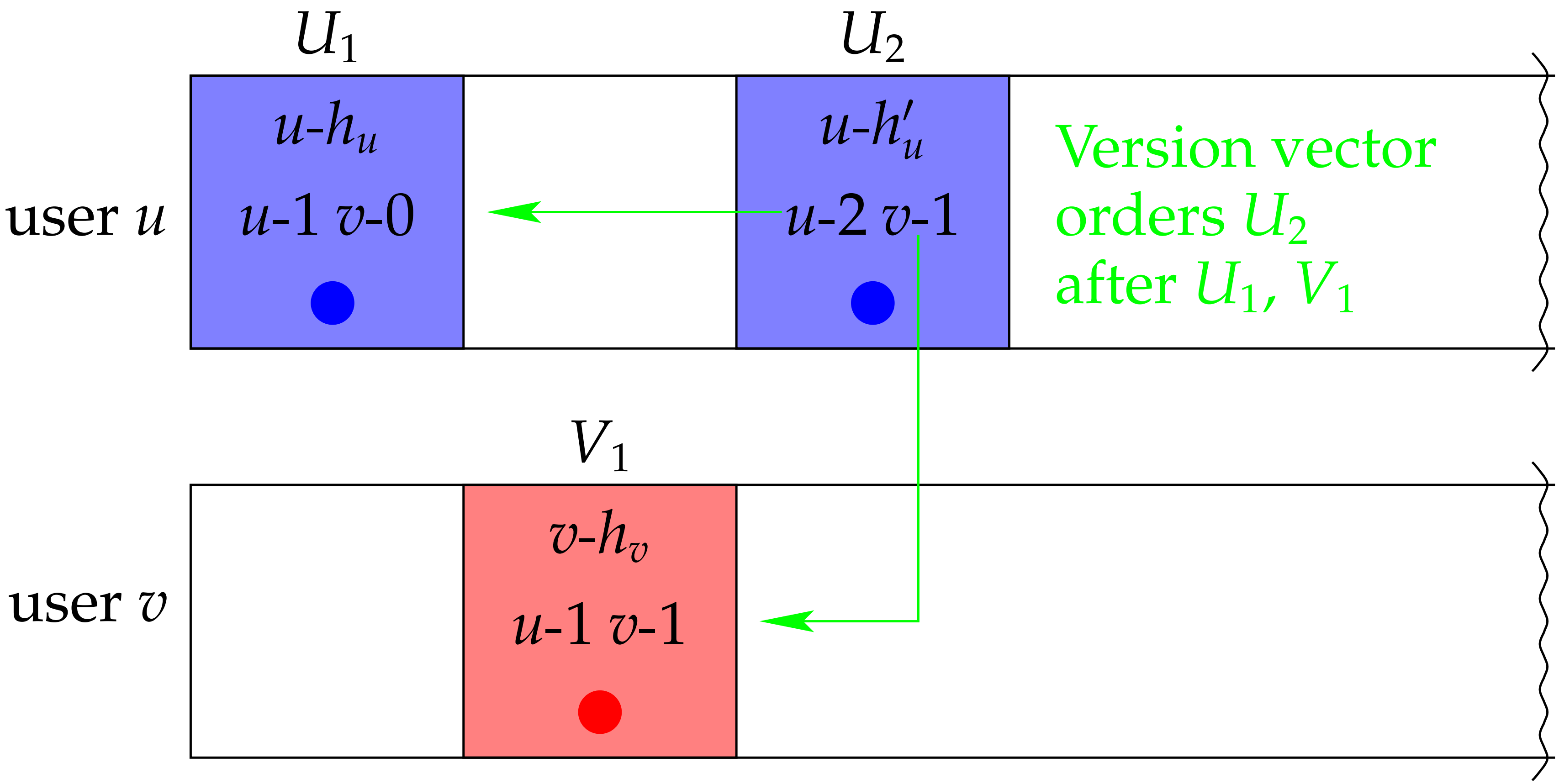
- **Still no concurrent updates**
- **Server maintains *version structure list* or VSL**
  - Contains latest version structure for each user/group
- **To fetch or modify a file, u's client makes 2 RPCs:**
  - PREPARE: Locks FS, returns VSL
  - Client sanity-checks VSL (ensures it is totally ordered)
  - 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:
    $\left\{u\text{-}h_u\ g\text{-}h_g, u\text{-}(n_u + 1)\ v\text{-}n_v\ \ldots\ g\text{-}(n_g + 1)\ \ldots\right\}_{K_u^{-1}}$
  - COMMIT: Uploads version struct for new VSL, releases lock

Example: Honest server



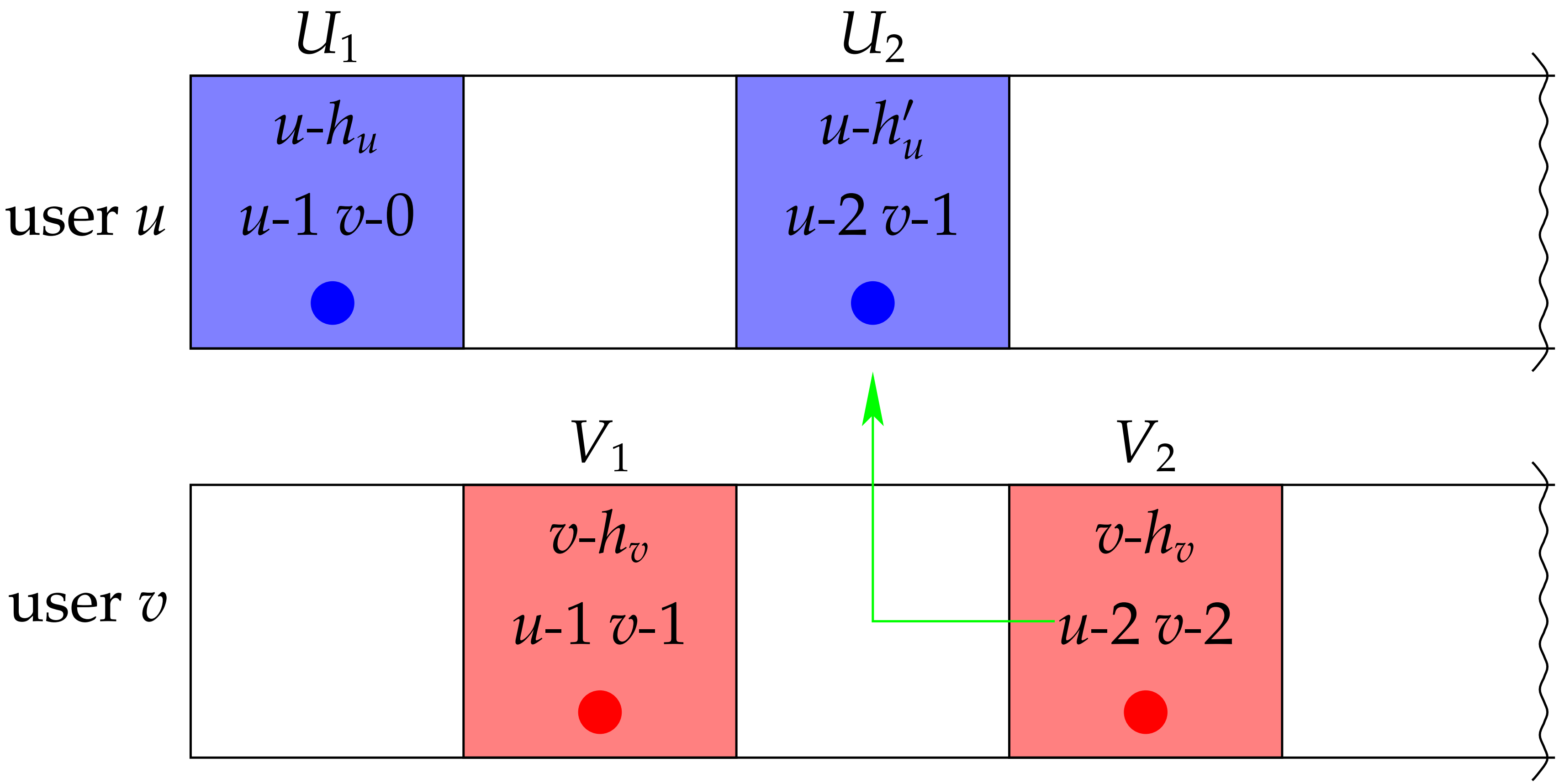
- **Users u and v each start at version 1 (sign U_1 & V_1)**

Example: Honest server



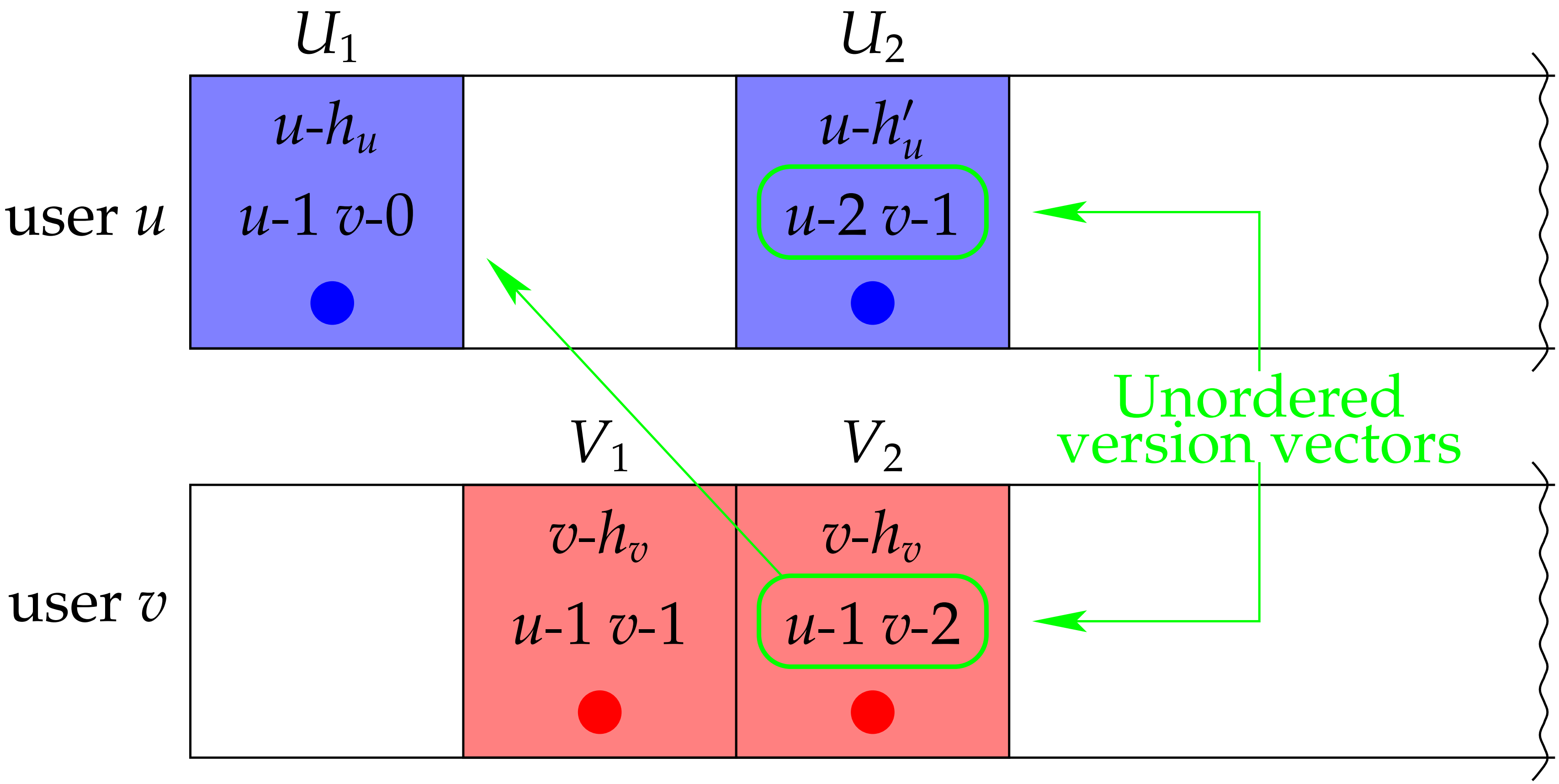
- **Users u and v each start at version 1 (sign U_1 & V_1)**
- **u modifies file f, signs U_2 w. new i-handle h'_u**

Example: Honest server



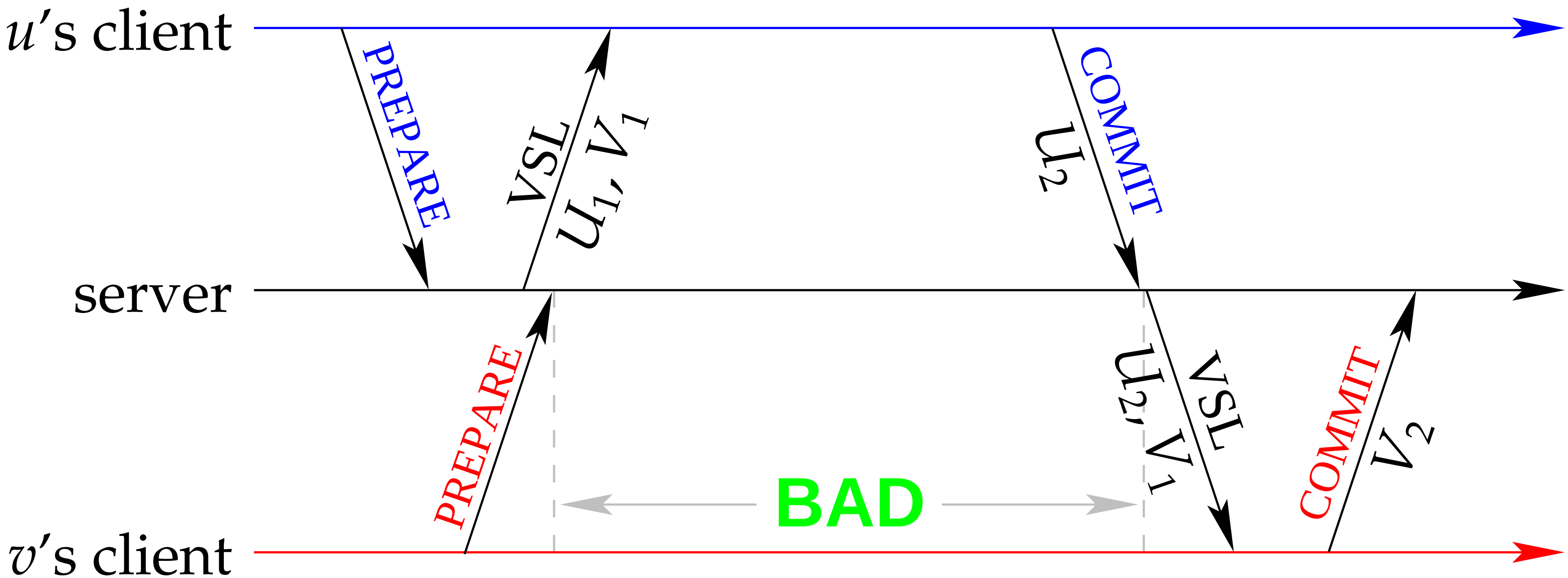
- **Users u and v each start at version 1 (sign U_1 & V_1)**
- **u modifies file f, signs U_2 w. new i-handle h'_u**
- **v fetches f, signs V_2 which reflects having seen U_2**

Example: Malicious server



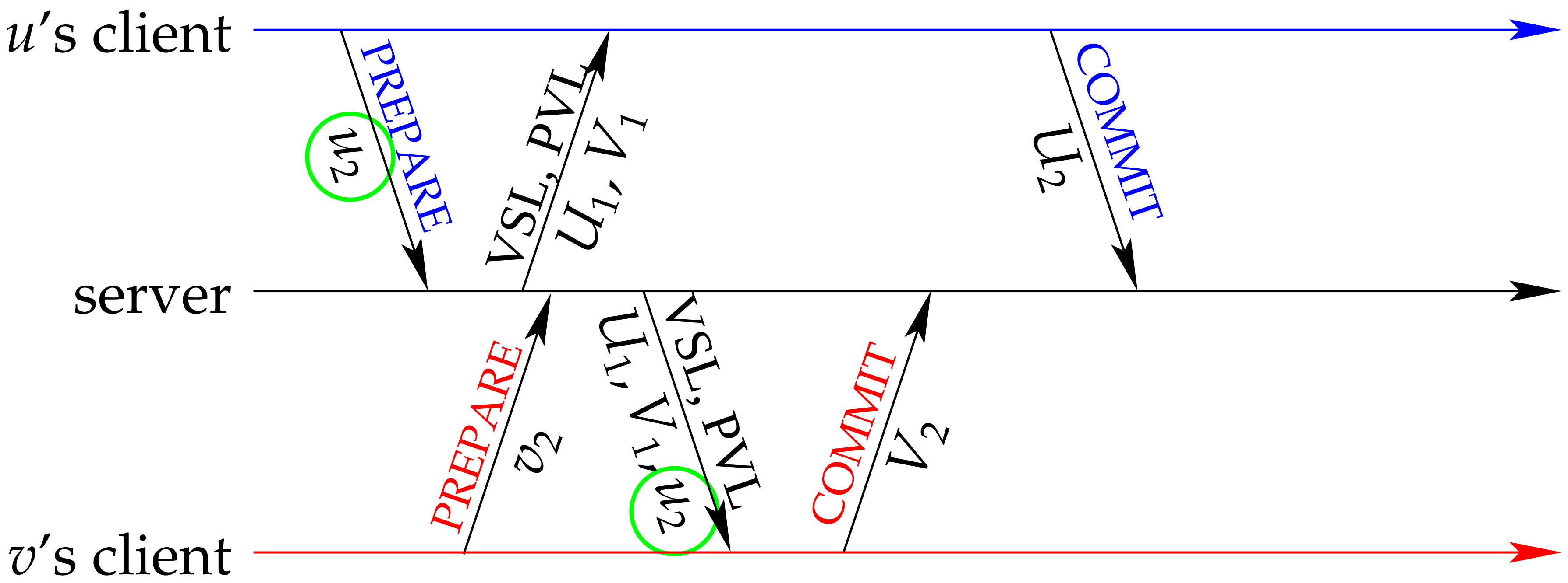
- **Suppose server hadn't shown u's modification of f to v**
- **Now $U_2 \not\leq V_2$ and $V_2 \not\leq U_2$**
  - u or v will detect attack upon seeing any future op by other

Limitations of serialized SUNDR



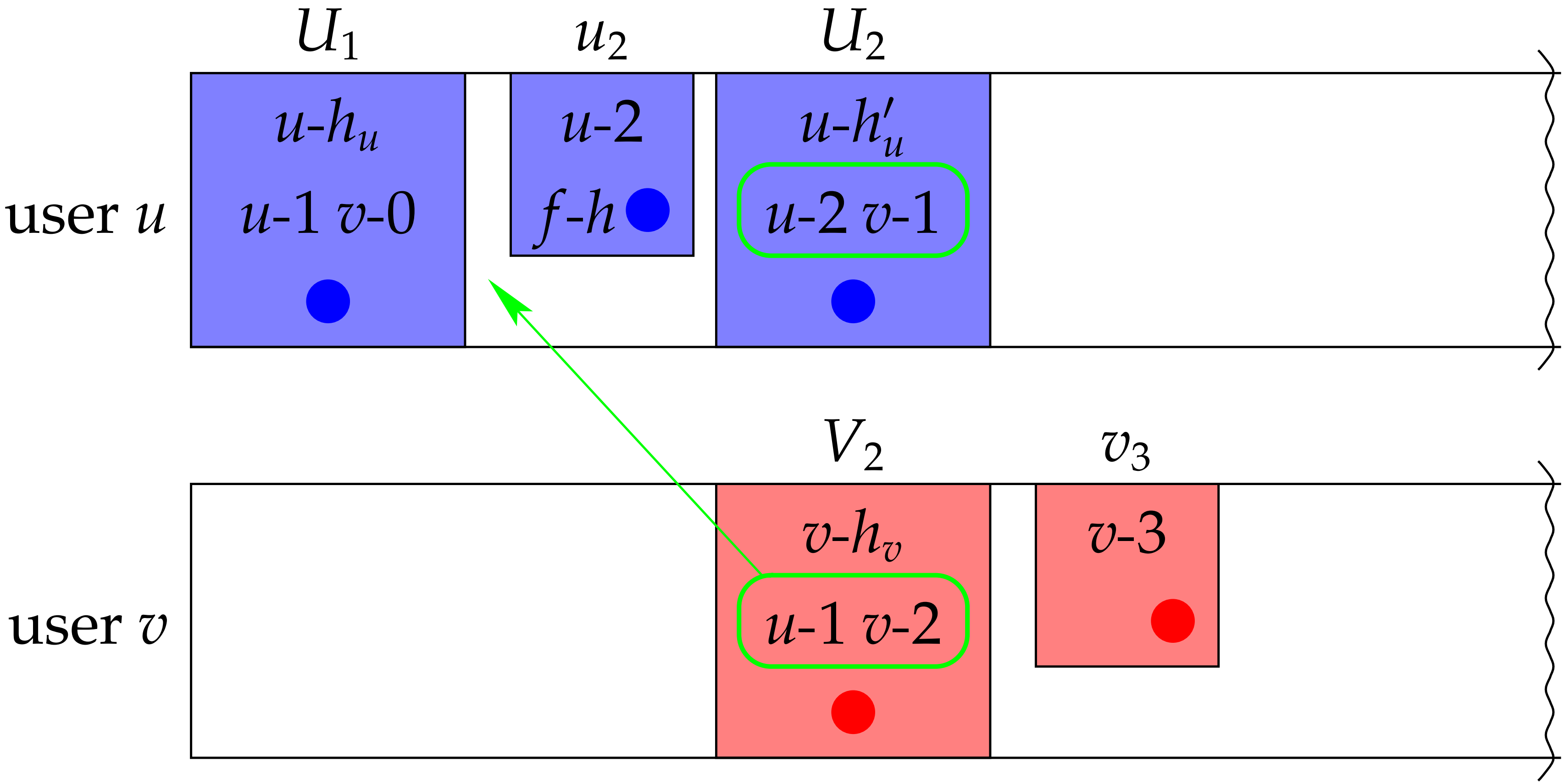
- **Honest server can only allow one operation at a time**
  - E.g., server must send U_2 to v to prevent fork on last slide
  - Must wait *even if V_2 doesn't observe any changes made in U_2*
- **Without concurrency, get terrible I/O throughput**

Solution 3: SUNDR



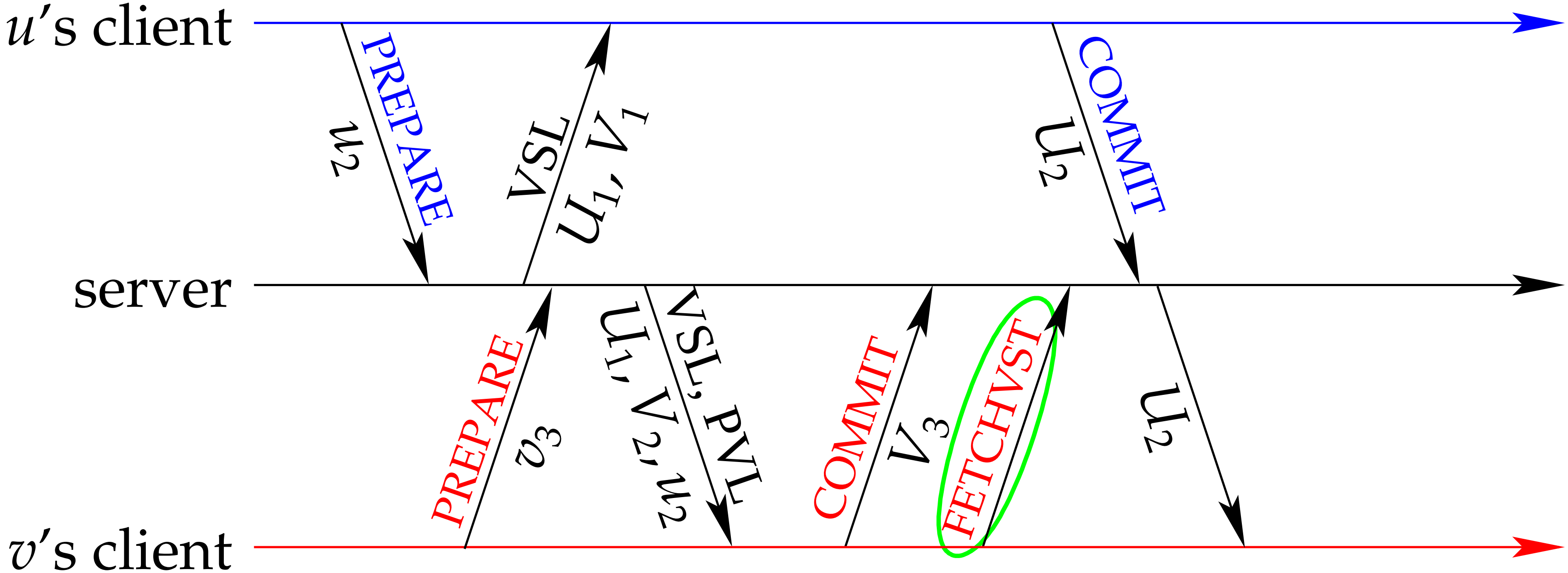
- **Pre-declare operations in signed *update certificates***
  - $u_2 = \{$"In vstruct U_2, I intend to change file f to hash h."$\}_{K_u^{-1}}$
- **Server keeps uncommitted update certificates in *Pending Version List* or PVL, returns with VSL**
- **Plan: Have v compute V_2 w/o seeing U_2 if it sees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

Danger: Erasing evidence of 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

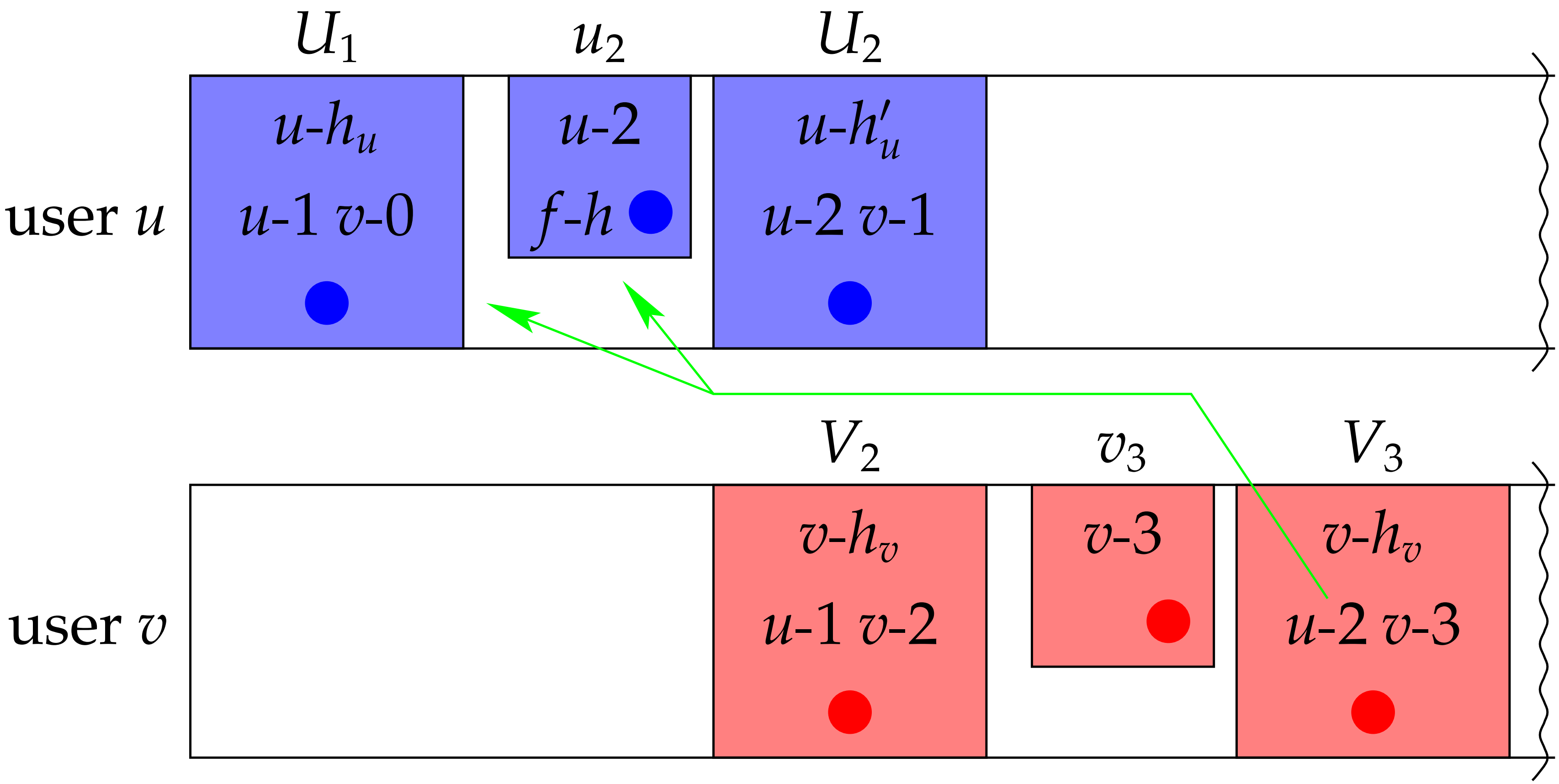
Case 1: Read-after write conflict



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

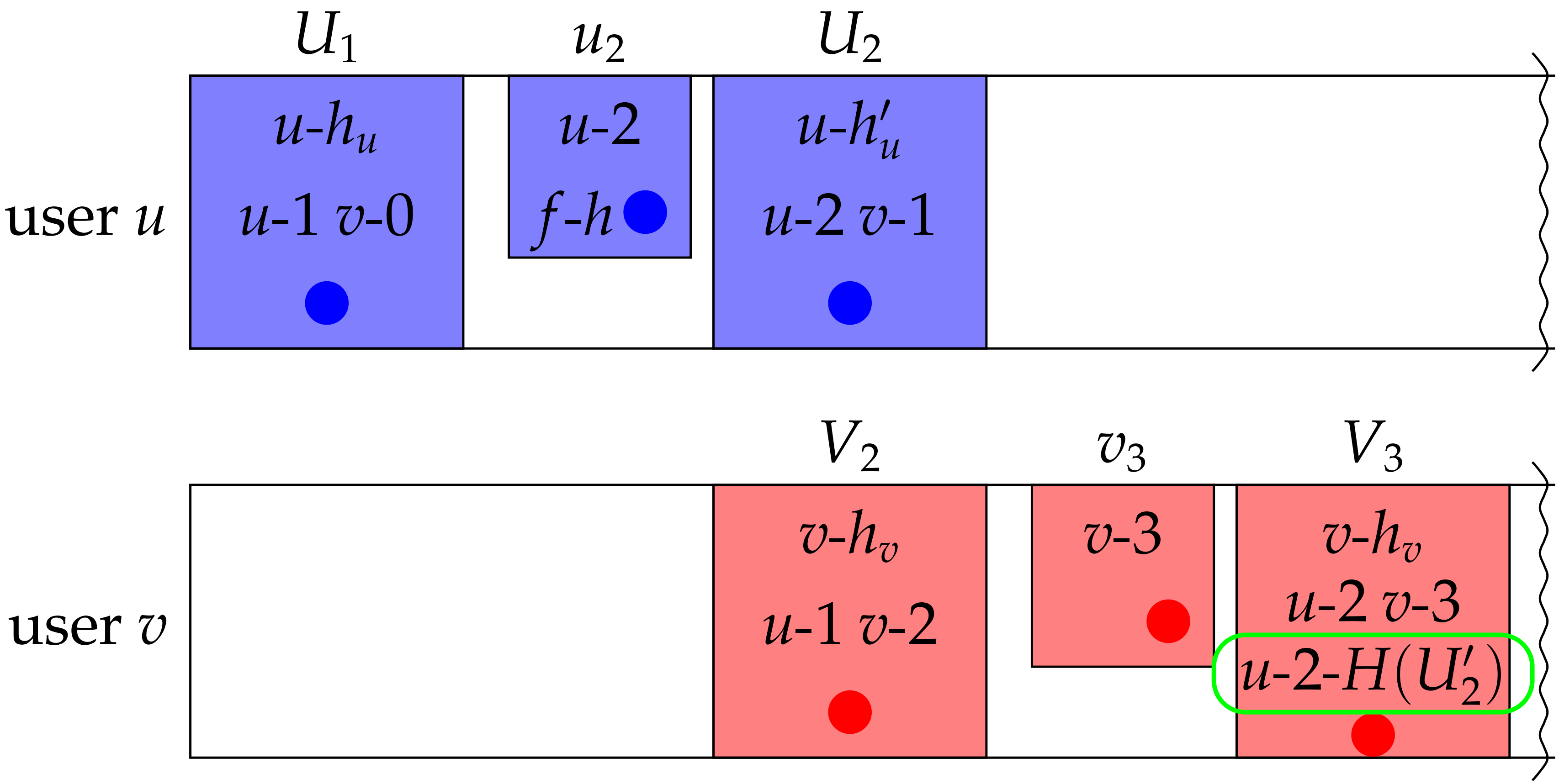
    v detects attack as $U_2 \not\preceq V_2$ (in VSL) and $V_2 \not\preceq U_2$

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

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 $U'_2 = \{u\text{-}2\ v\text{-}2\} \neq U_2$

Concurrent version structures

- **Define collision-resistant hash V for vstructs**
  - E.g., delete i-handle, sort u-n/u-n-h data, run through H
- **Version structures now reflect pending updates**

$$\{\underbrace{u\text{-}h_u\ g\text{-}h_g,}_{\text{i-handles}}\ \underbrace{u\text{-}4\ \ v\text{-}3\ \ \ldots,}_{\text{version vector}}\ \underbrace{v\text{-}3\text{-}k\ u\text{-}4\text{-}\perp\ \ \ldots}_{\text{pending}}\}_{K_u^{-1}}$$

  - Vstruct has a u-n-k triple for each PVL entry
  - $u, n =$ user,version of a pending update
  - k is V of a version structure, or reserved “self” value $\perp$
- **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 $= \perp$

Ordering concurrent vstructs

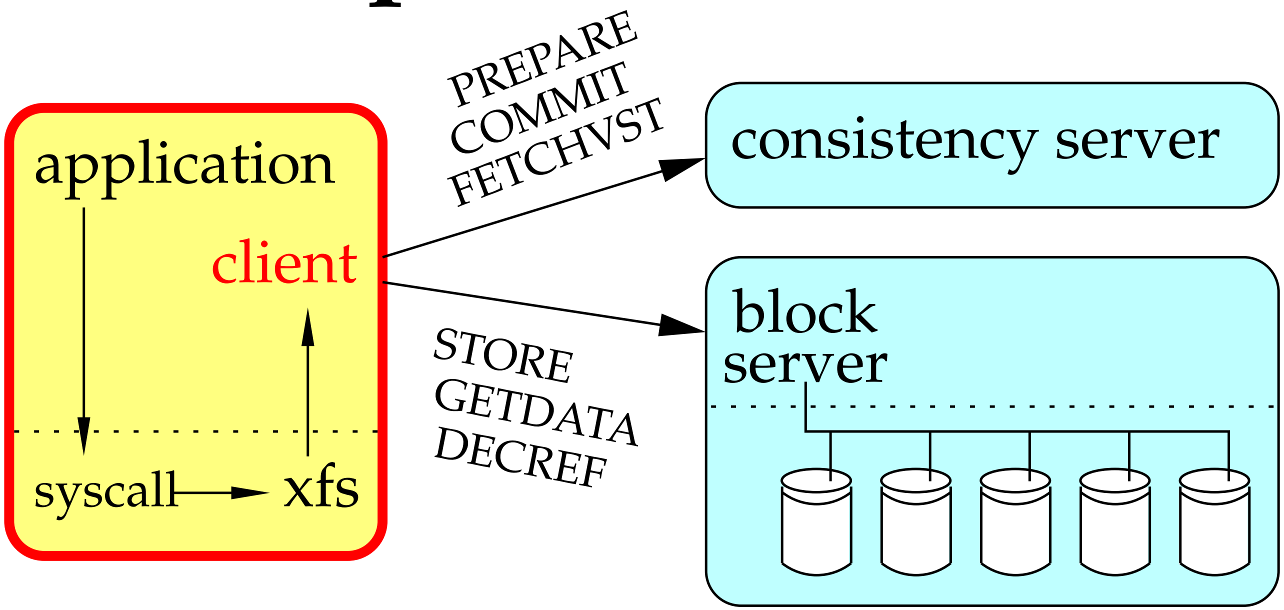
Definition. We say $x \leq y$ iff:

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-k in y, one of the following conditions must hold:
   (a) $x[u] < n$ (x happened before the pending operation that u-n-k represents), or
   (b) x also contains u-n-k (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).

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

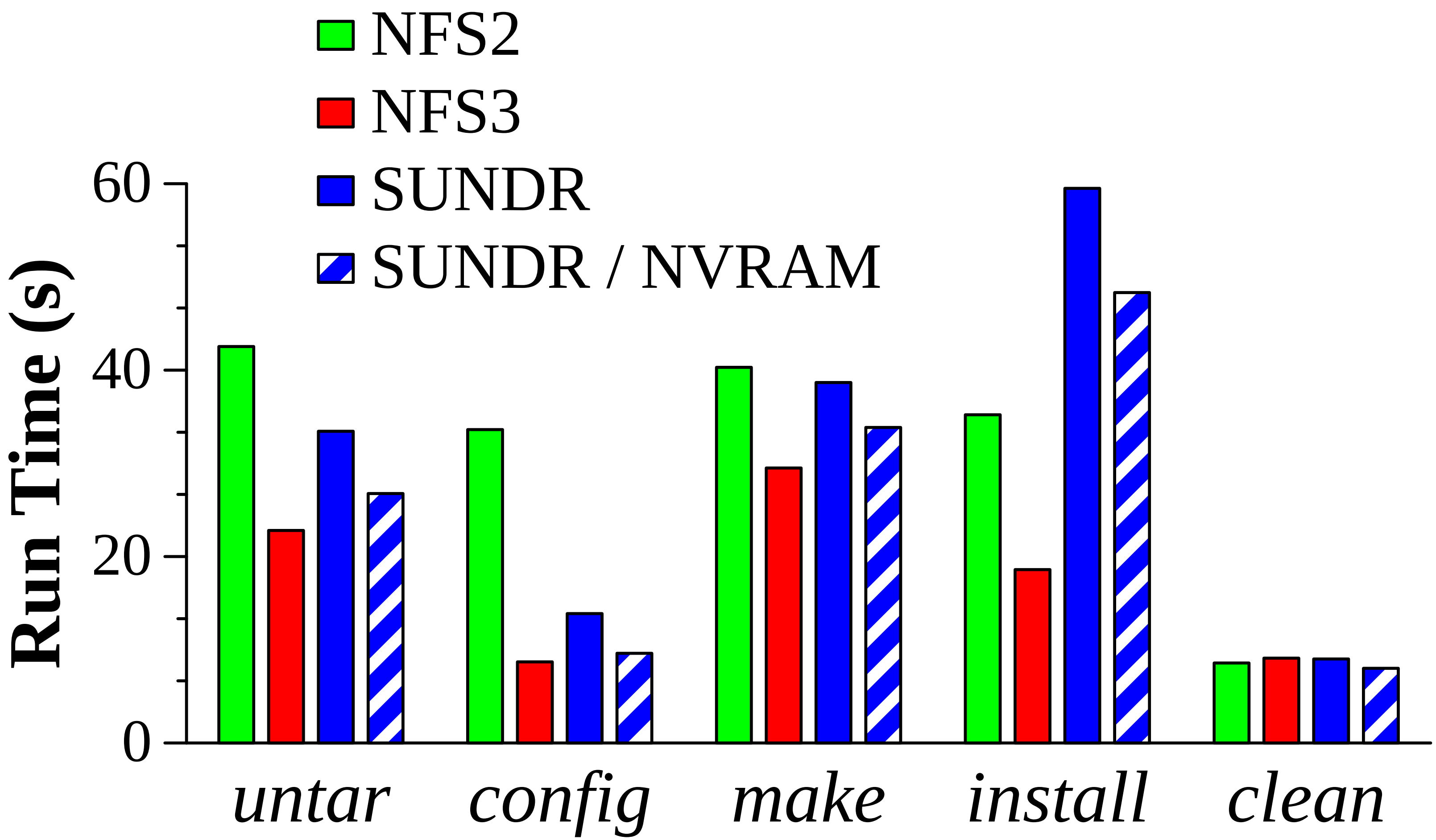


- **Client based on xfs device driver**
  - xfs part of Arla, a free AFS implementation
  - Designed for AFS-like semantics
- **Server split into two daemons**
  - *Consistency server* handles update certs, version structs
  - *Block server* stores bulk of data
  - Can run on same or different machines

Further optimizations

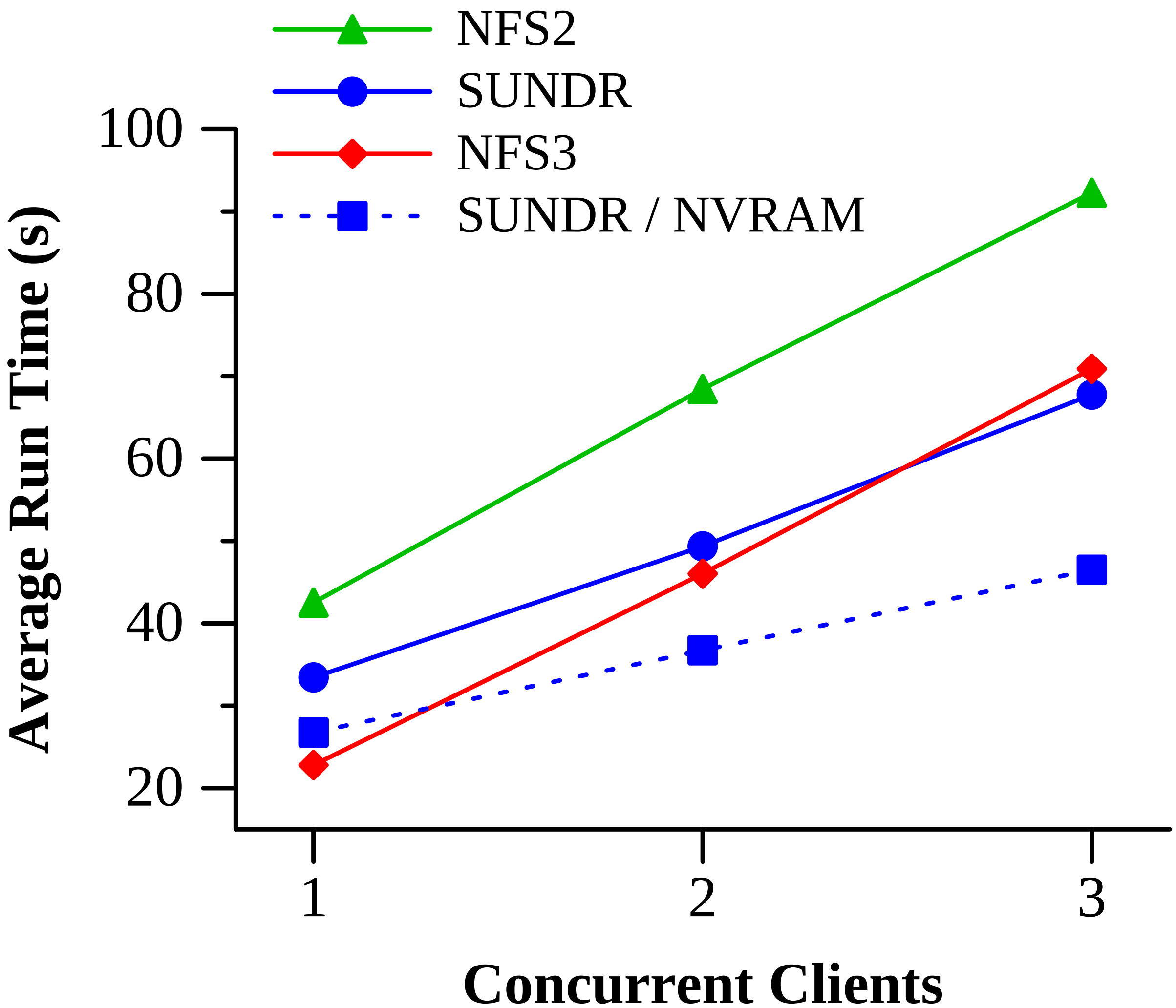
- **i-handles really hash plus some deltas**
  - Amortizes recomputing hash tree over multiple ops
- **Include multiple fetches/modifies in one operation**
- **i-tables are Merkle B+-trees**
- **Group i-tables add yet another level of indirection**
  - No need to change group i-table if same user writes 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 "$\leq$"

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

Scalability to multiple clients



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