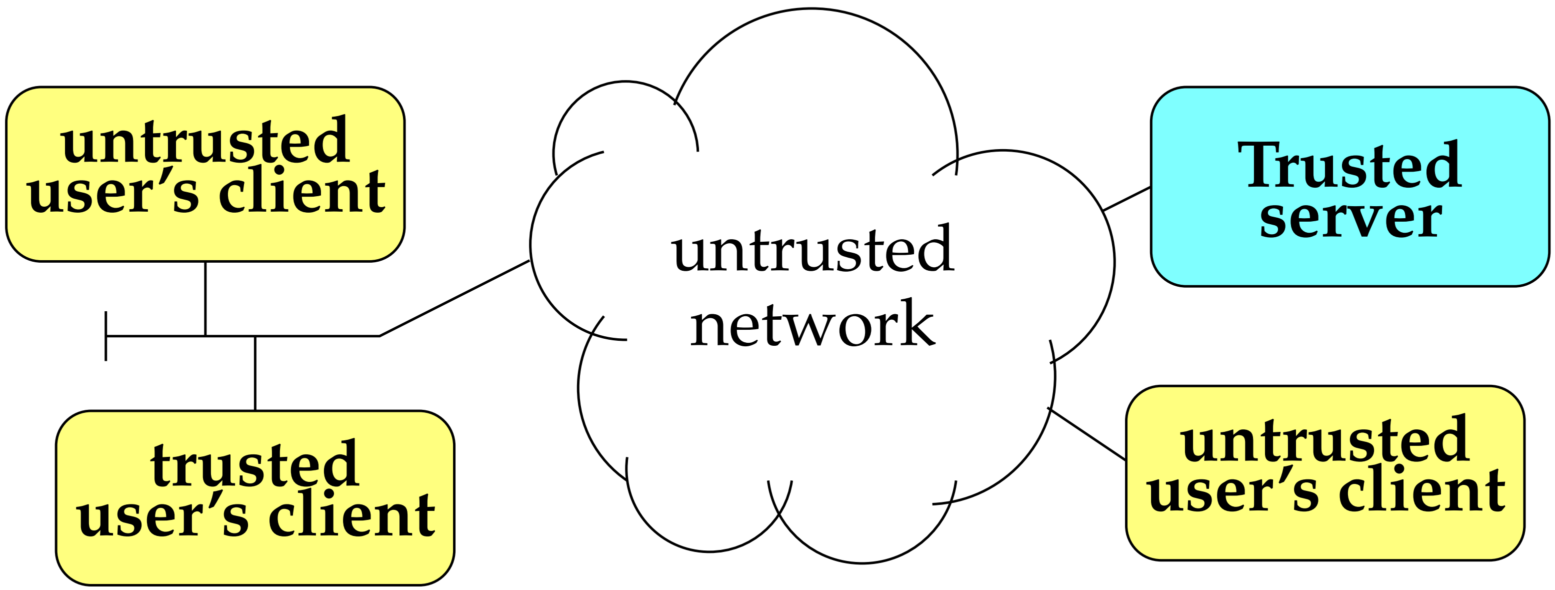
Traditional network file system security

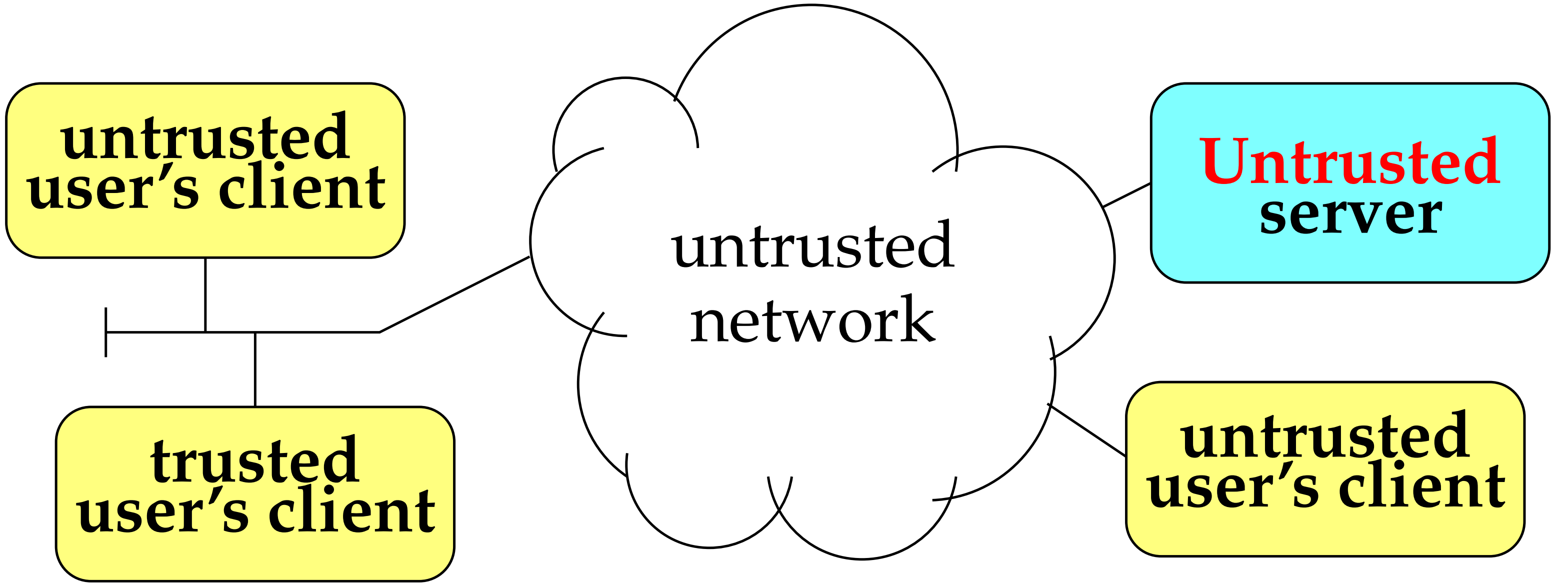


- All communications mutually authenticated
- Server trusted to reflect only authorized modifications

The problem: Trusting the server

- **People with server access shouldn't have data access**
  - System administrators, contractors, server collocation sites, data warehouses, web/file hosting servivces, ...
  - Network attackers often gain complete access to servers
- **Anyone with server access can tamper with data**
- **Yet people expect fail-stop behavior from servers**
  - Server may crash; people will recover with backups
  - No protection against unauthorized data modification
- **No system has achieved anything like traditional network FS semantics without trusting the storage.**

The SUNDR file service



- **A provably secure file system protocol**
  - Secure whether or not the server obeys the protocol
- **A notion of file system consistency – fork consistency**
  - Server may fail, but only in readily detectable ways
- **Implementation underway at NYU**

SUNDR approach

- **Give every user & server a public signature key**
  - Assume for talk that all parties know each others' keys
    (Can use the file system to manage the keys)
- **Users sign state of file system on every operation**
  - Clients get state of file system from signed data
  - Compare users' signed data for consistency
  - Assumes signatures are cheaper than network RTT
    (Increasingly valid assumption as CPUs improve)

Threats from a malicious file server

0. **Violation of Secrecy (not addressed in this talk)**
1. **Data forgery**
   - Attacker can substitute arbitrary data for a file
   - Attacker can substitute random/specific data for a file
2. **Data freshness attacks**
   - Attacker can roll back any file/block to a previous state
3. **Forking attacks – complete partitioning of users**
   - If you see a user's activity, all is well. However, server might *fork* users, hiding their actions from each other
4. **Denial of service – readily detectable**

SUNDR Goal: Prevent 1–2, facilitate detection of 3

Related work: Cryptographic storage

- **Encrypt all files on disk**
  - Attacker cannot read encrypted files
  - Tampering with data produces garbage
- **Does not ensure integrity or freshness**
  - Inserting garbage in files may be useful attack
  - Attackers can roll back file contents to previous version
  - Anyone with read access can forge a file's contents
- **Many files more widely readable than writable**
  - Challenge: Sharing files some can write and others can't
  - Need digital signatures for untrusted users to verify files

Traditional file system semantics

- **One often hears of "close-to-open consistency"**
  - User A writes and closes a file f on one client
  - User B subsequently opens f on another client
  - B should read the contents written by A
  - Close-to-open a misnomer – e.g., truncate w/o open/close
- **Instead, let's speak of *fetch-modify consistency*.**
  - 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, ...
- **View FS as clients performing fetch/modify ops**

Ordering of file system operations

Definition. A set of fetch and modify operations is **ordered** iff:

- Every op has wall-clock *issue* and *completion* times (for model only – client & server don't know times)
- Every op's completion time is after its issue time
- There is a partial order, *happens before* ($\prec$), such that:
  - If O_1 completed before O_2 issued, then $O_1 \prec O_2$
  - $\prec$ orders any two operations by the same client
  - $\prec$ orders any two conflicting operations (i.e., a modification and any other op on same file)

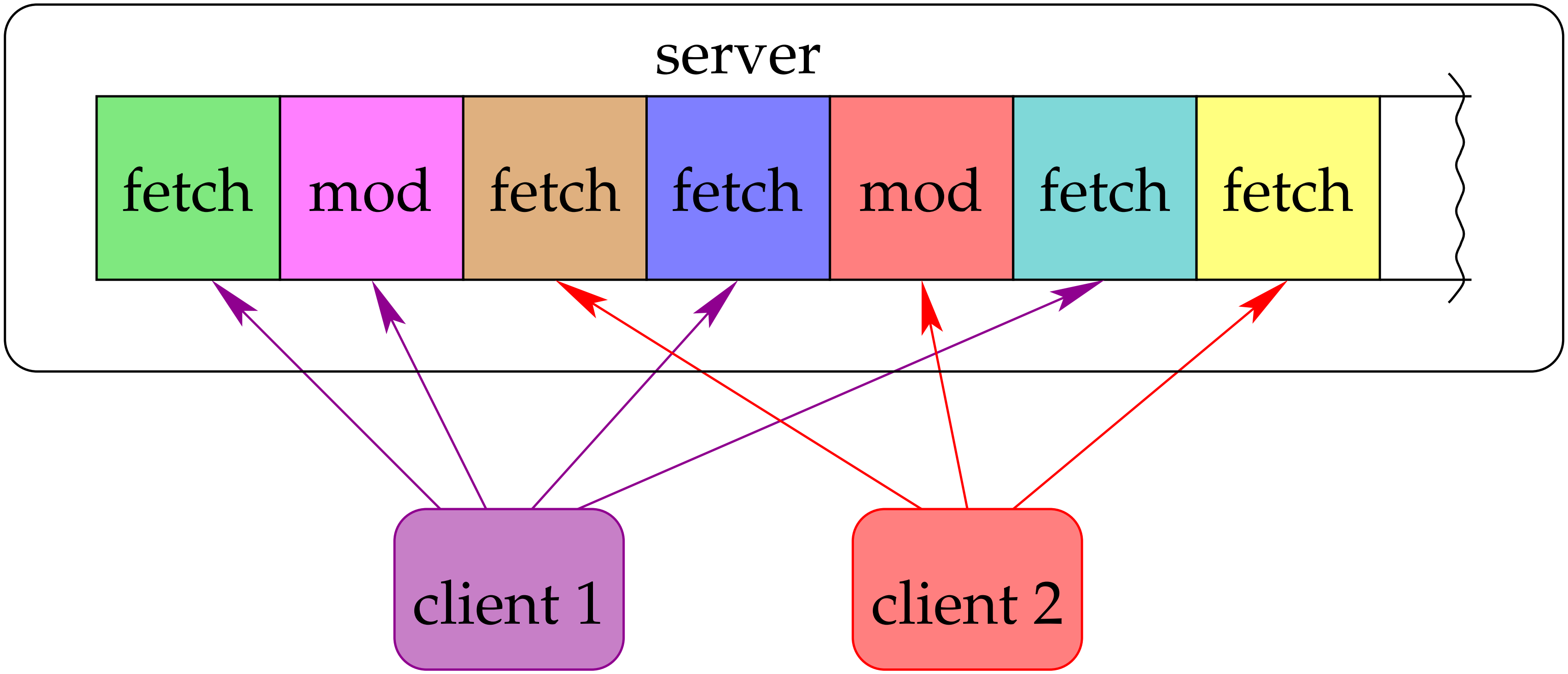
Fetch-modify consistency

Definition. A set $\mathcal{O}$ of fetch & modify operations is **fetch-modify consistent** iff $\mathcal{O}$ is ordered and any fetch F of a file p reflects exactly the modifications of p that happened before F.

Question: How close can we get to fetch-modify consistency without trusting the server?

Answer: Fork consistency

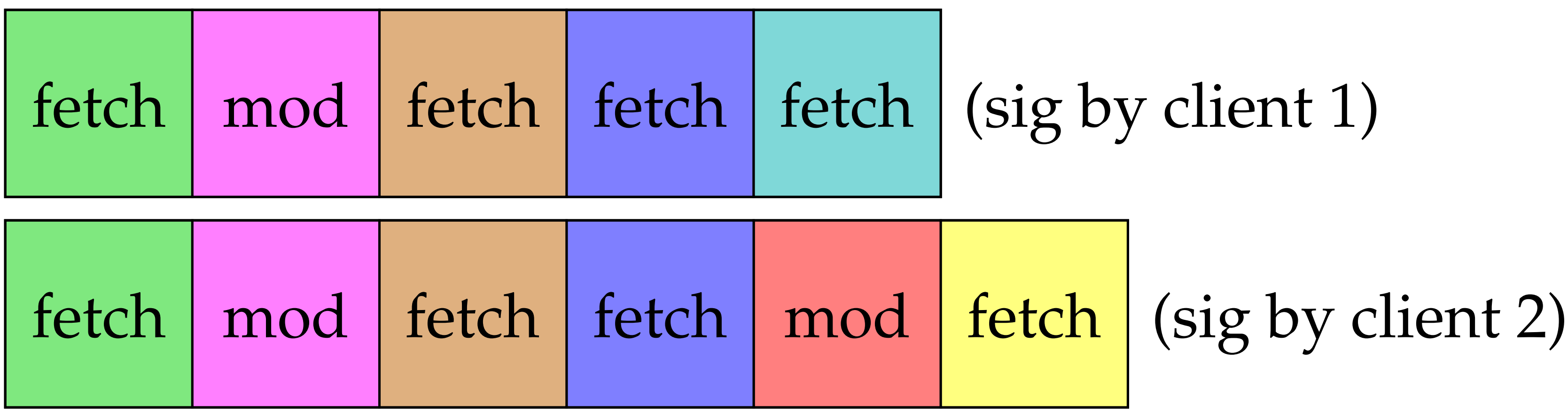
Straw-man: Signed history



- **Server keeps complete log of all operations**
  - No concurrent operations (untrusted lock serializes all ops)
- **Each log entry signed by principal performing op**
  - Signature covers current operation + entire past history

What can a malicious server do?

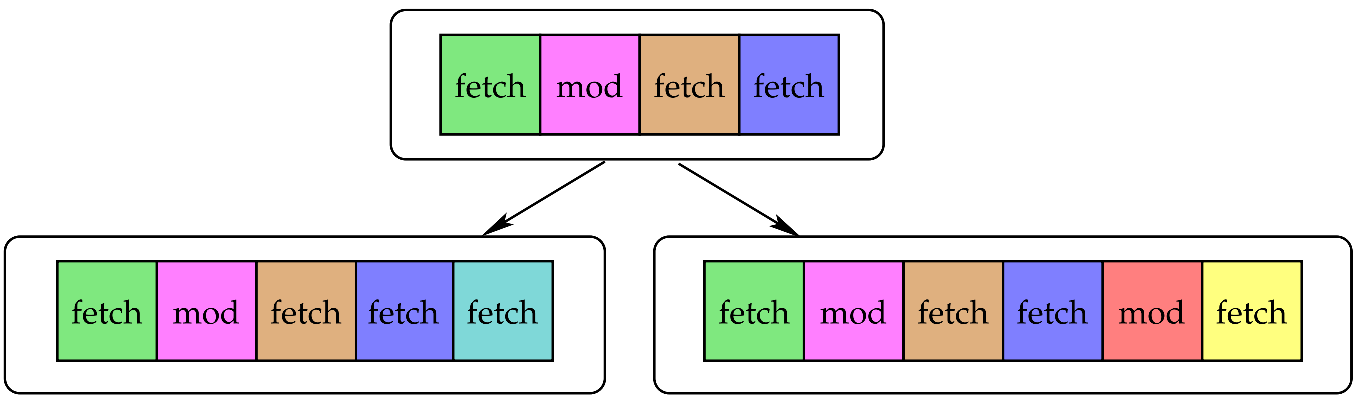
- **Clients verify signatures on log entries**
  - Prevents data forgery attacks
- **Clients check compatibility of signed histories**
  - Check any two histories by ensuring one is prefix of other
  - Prevents data freshness attacks
- **Consistency violations produce incompatible histories:**



- **Detected if ever one client sees other's later history**

Fork consistency

- **Consider the following set of histories:**
  - *Maximal* signed histories (that are not prefixes of others)
  - The greatest common prefix of every two maximal histories
- **Arrange histories as a graph**
  - Put an edge to each node from its longest prefix:



- **Histories will form a tree**
  - Once forked, two users can never be joined (see same op)
  - Thus, we call this property **fork consistency**

Fork consistency formalized

Definition. Let $\mathcal{O}$ be a set of completed operations. A **forking tree** on $\mathcal{O}$ is a tree, each node of which has a subset of $\mathcal{O}$ called a **forking group**, such that:

- Each forking group is fetch-modify consistent
- For any client c, at least one f.g. has all c's operations
- Any op occurs in a highest node n + all descendents of n
- If $O_1 \prec O_2$ in g_1 and $\{O_1, O_2\} \subseteq g_2$ then $O_1 \prec O_2$ in g_2
- $\forall O \in g$, either $O \in \mathrm{parent}(g)$ or $\forall O' \in \mathrm{parent}(g),\ O' \prec O$

Definition. A file system is **fork consistent** iff it there always exists a forking tree on all completed operations.

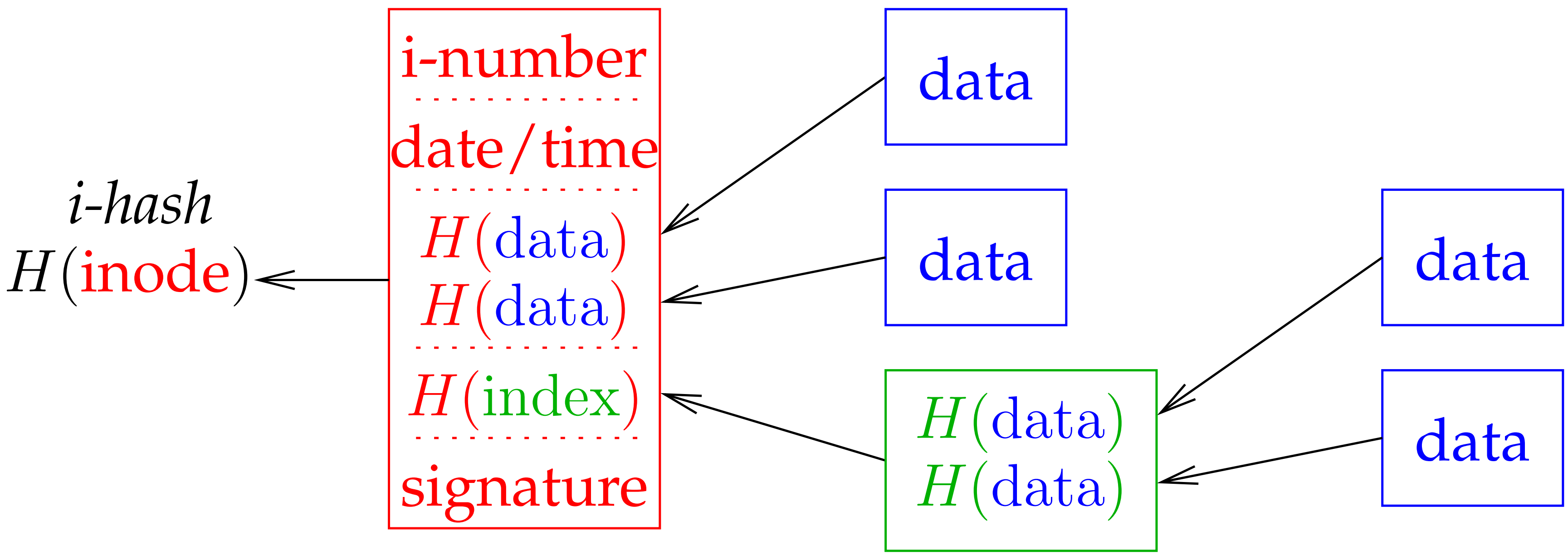
Implications of fork consistency

- **Magnifies subtle consistency failures**
  - Two users see all of one another's changes or none
  - A *fork attack* partitions users into disjoint sets
  - Users who communicate will easily notice problem
  - Users who log into same client will easily notice problem
- **Can trivially audit server retroactively**
  - If you see effects of operation X, guarantees file system was consistent at least until X was performed
  - Clients that communicate get fetch-modify consistency
    E.g., two clients on an Ethernet when server "outsourced"
  - Exchange information about a recently modified file
  - Pre-arrange for "timestamp" box to update FS every minute

Implementing fork consistency

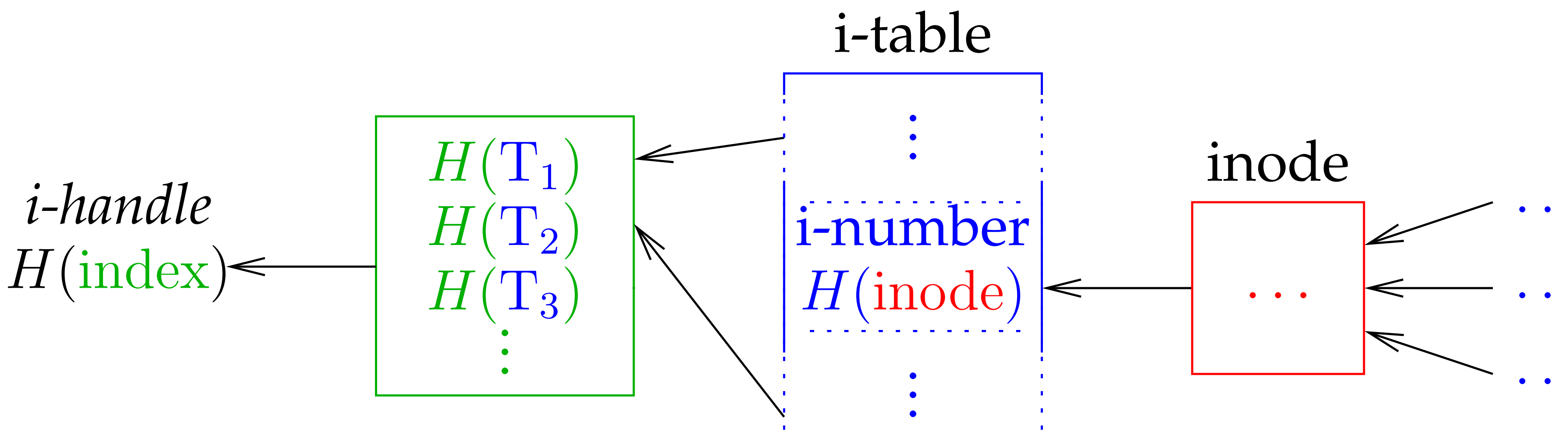
- **Keeping complete file system history not practical**
  - Would need to garbage collect at some point
  - Signing large histories would be expensive
- **Instead, use a collision-resistant hash function H (Intractable to find $x \neq y,\ H(x) = H(y)$)**
  - All files writable by a user or group are specified by a short (20-byte) **i-handle**
- **SUNDR then takes a two-pronged approach:**
  - Block protocol verifies file data based on i-hendles
  - Consistency protocol handles fetch/mod of i-handles

Compressing files into handles



- **Hash file data blocks**
- **Store hashes in indirect blocks & i-nodes**
- **Hash i-nodes to get i-hashes**
  - Given i-hash, can verify any block of file

Compressing i-hashes into i-handles



- **Build per-usr/grp i-table mapping i-number → i-hash**
  - Directories map filename → ⟨user/group, i-number⟩
- **Hash tree compresses i-table into i-handle**
- **Each user/group digitally signs its own i-handle**

Implementing a consistent file system

- **Easy *if* clients can get latest i-handles**
- **To *fetch* a file:**
  - Fetch latest i-handle
  - Retrieve any i-table, i-node, and data blocks not in cache
- **To *modify* a file**
  - Store new blocks on server
  - Sign new i-handle
  - Make new i-handle available to other users

The SUNDR block protocol

- **User and server authentication (straight-forward)**
- **STORE (*block*) – store *block*/bump per-user refcnt**
- **RETRIEVE (*hash*) – retrieve block with *hash***
- **UNREF (*hash*) – decrement per-user refcnt**
- **UPDATE (*certificate*) – get all i-handles**
- **COMMIT (*version info*) – commit new i-handle**
- **Crash recovery functions**

Implementing i-handle consistency

- **Users assign increasing vers. nos. to their i-handles**
- **Idea: Users sign each other's version numbers:**
  - Each user u_i maintains a *version structure*:
    $\{\text{VRS}, \text{principal-ihandle}, u_1\text{-}n_1\ u_2\text{-}n_2\ \dots\ u_i\text{-}n_i\ \dots\}$
  - When updating its i-handle, a user bumps its own version
    $\{\text{VRS}, u_i\text{-}h, u_1\text{-}n_1\ u_2\text{-}n_2\ \dots\ u_i\text{-}(n_i+1)\ \dots\}_{K_{u_i}^{-1}}$
  - When updating a group, a user bumps his & group's no.:
    $\{\text{VRS}, u_i\text{-}h\ g\text{-}h_g, u_1\text{-}n_1\ u_2\text{-}n_2\ \dots\ g\text{-}(n_g+1)\ \dots\ u_i\text{-}(n_i+1)\ \dots\}_{K_{u_i}^{-1}}$
- **All signed version structures must be ordered**
  - Let $y[u]$ be u's version in y, or 0 if u not in y
  - Say $x \leq y$ iff $\forall u\ x[u] \leq y[u]$
  - Two unordered structures indicate a forking attack

A "bare-bones" protocol

- **Simplify the problem for bare-bones protocol:**
  - Still no concurrent updates (assume untrusted lock)
- **Server maintains users' latest signed i-handles in *version structure list* or VSL.**
- **To fetch or modify a file, u_i's client makes 2 RPCs:**
  - UPDATE: Locks FS, downloads and sanity-checks VSL
  - Calculates & signs new version structure:
    $\{\text{VRS}, u_i\text{-}h, u_1\text{-}n_1\ u_2\text{-}n_2\ \ldots\ u_i\text{-}n_i\ \ldots\}_{K_{u_i}^{-1}}$
  - COMMIT: Uploads version struct for new VSL, releases lock

Example

Users u and v both start at version 1:

$$\begin{aligned} y_u &= \{\text{VRS}, u\text{-}h_u, u, u\text{-}1 \ \ldots\}_{K_u^{-1}} \\ y_v &= \{\text{VRS}, v\text{-}h_v, v, u\text{-}1\ v\text{-}1 \ \ldots\}_{K_v^{-1}} \end{aligned}$$

u updates a file, and bumps version number to 2:

$$\begin{aligned} y_u &= \{\text{VRS}, u\text{-}h'_u, u\text{-}2\ v\text{-}1 \ \ldots\}_{K_u^{-1}} \\ y_v &= \{\text{VRS}, v\text{-}h_v, u\text{-}1\ v\text{-}1 \ \ldots\}_{K_v^{-1}} \end{aligned}$$

v fetches the file, bumps its version number, reflects u-2:

$$\begin{aligned} y_u &= \{\text{VRS}, u\text{-}h'_u, u\text{-}2\ v\text{-}1 \ \ldots\}_{K_u^{-1}} \\ y_v &= \{\text{VRS}, v\text{-}h_v, u\text{-}2\ v\text{-}2 \ \ldots\}_{K_v^{-1}} \end{aligned}$$

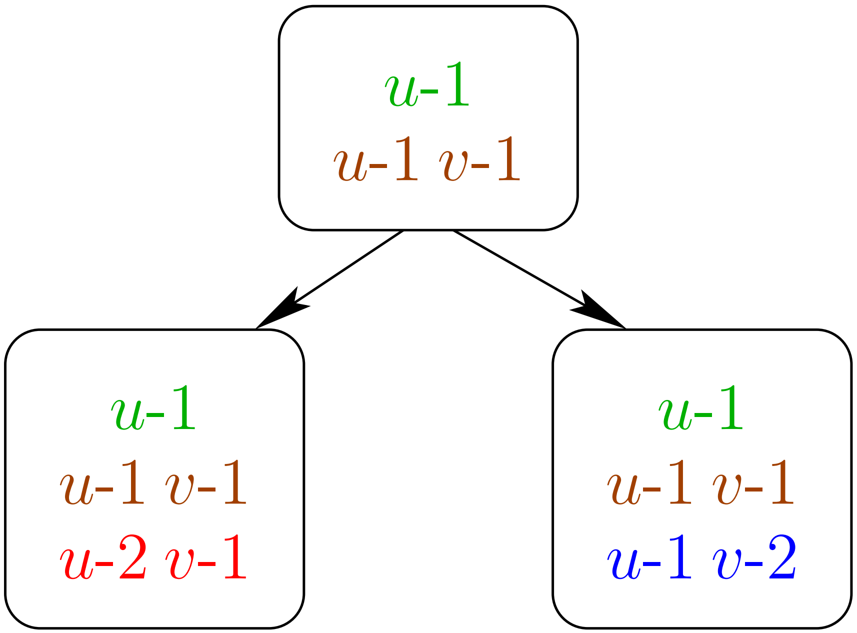
Attack

Suppose v hadn't seen u's latest i-handle h', then:

$$y_u = \{\text{VRS}, u\text{-}h'_u, u\text{-}2\ v\text{-}1\ \ldots\}_{K_u^{-1}}$$

$$y_v = \{\text{VRS}, v\text{-}h_v, u\text{-}1\ v\text{-}2\ \ldots\}_{K_v^{-1}}$$

Now $y_u \not\leq y_v$ and $y_v \not\leq y_u$. u and v can never see one another's updates again (partitioned). Forking tree:



Protocol correctness theorem

Theorem: A set of (completed) operations on a file system is fork consistent if there exists a partial order $<$ on operations with the following two properties:

1. Every two distinct operations created by a single client are ordered by $<$.
2. For any operation q, the set $\{o \mid o \leq q\}$ of all operations (by any client) less than or equal to q is totally ordered and fetch-modify consistent with $<$ as the happens-before relation.

Allowing concurrent updates

- **Bad to lock FS between UPDATE & COMMIT**
- **Fix1: pre-declare operations in UPDATE certificates**
  $\{\textbf{UPDATE}, u, n+1, H(y_u), [\langle \textbf{usr/grp}, \textbf{inum}, \textbf{ihash}\rangle, \ldots]\}_{K_{u_i}^{-1}}$
  - Specify new version number, hash of old version struct
  - Specify new i-hashes for any modified files (deltas for dirs)
- **Fix2: reflect pending ops in version structures**
  - Fold any pending modifications into new i-handles
  - List COMMITs not available at tine of signature
  - Conflicting fetches won't complete before COMMIT

Concurrent protocol

- **Server keeps list of pending updates in *pending version list* or PVL**
  - Contains signed update certificates
  - Contains future version structs, unsigned & w/o i-handles (Server can calculate vers structs as it determines order)
  - Server replies to UPDATE with both VSL and PVL
- **Concurrent clients must only wait if conflict:**
  - When opening an updated file, wait for commit
  - Otherwise, can tell no conflict, so proceed immediately
  - i-hashes let user recover if client crashes

Concurrent protocol details

- **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 hash of a version structure, or reserved "self" value $\perp$ (u's nth version structure always contains u-n-$\perp$)

- **Define collision-resistant hash V for version structs**
  - E.g., delete i-handle, sort u-n/u-n-h data, run through H

- **PVL contains future version structures**
  - Each entry is of the form $\langle$update cert, $\ell\rangle$
  - ℓ is unsigned version structure to be, but i-handle $= \perp$
  - Clients compute each u-n-h triple with $V(\ell)$

Ordering concurrent version structures

Definition. We now 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).

Informal justification

- **If $x \leq y$:**
  - y must reflect any operations that were pending when x signed. This follows from $x[u] \leq y[u]$ for all u, since pending versions numbers are reflected in version structure.
  - For operation o pending when y was signed: Either x reflects o was pending, or x "happened before" o.
- **If client saw operation o committed when it signed x, any version structure greater than x must also be signed by someone who saw o committed.**

Future work

- **Low bandwidth file system protocol**
  - Because SUNDR based on hashing, ideal for LBFS technique [SOSP'01]
- **High-performance log-structured server**
- **Combine with archival storage**
  - Venti [FAST'01] suggests keeping all unique hashed blocks practical
- **Untrusted peer-to-peer file cache**
  - Don't trust server anyway
  - Might as well get data from untrusted peer
- **Data secrecy (cryptographic storage)**

Summary

- **Eliminate trust in network file servers**
  - Administrative issues shouldn't drive security policy
  - Make servers far more immune to network attacks
- **Fork consistency makes server failures detectable**
  - Most server failures immediately detected
  - Only complete partitioning of users may go undetected
  - But users can easily check this in a variety of ways
- **Fork consistency is practical w/o trusted server**
  - Two signatures + $1\frac{1}{2}$ round trips per FS operation