Building Secure File Systems out of Byzantine Storage

David Mazières
joint work with
Jinyuan Li (implementation) and Dennis Shasha
NYU Department of Computer Science

Motivation

- **Many people have access to data who don't need it**
  - System administrators, contractors, server collocation sites, data warehouses, web/file hosting servivces, ...
  - Server access driven by administrative needs, not security
- **Servers are attractive targets for network attacks**
- **People expect fail-stop behavior from servers**
  - Server may crash; people will recover with backups
  - But what about subtle, undetected tampering (e.g., rootkit)?
  - Backups won't help with a failure you don't know about
- **No system has achieved anything like traditional network FS semantics without trusting the storage.**

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, ...
  - For the rest of talk, will assume some intuitive mapping

File system model

Definition. A **principal** is an entity authorized to access the file system.

Definition. A **client** produces a series of fetch and modify requests. Each request has a wall-clock *issue time*.

Each request is on behalf of a principal.

The client sends its requests to the server.

We call requests processed by the server "operations."

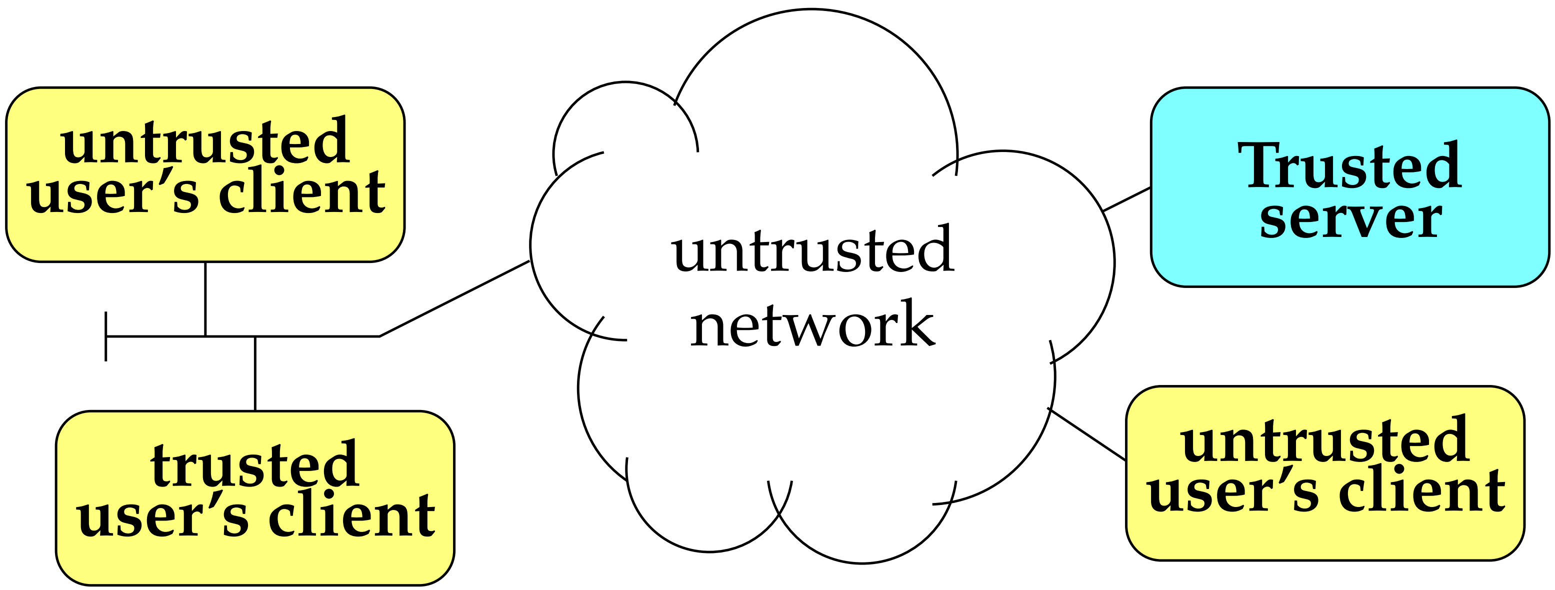
Formal fetch-modify consistency

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

- Each operation has a *completion time* (after issue)
- 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 a mod. wrt. all other ops on same file

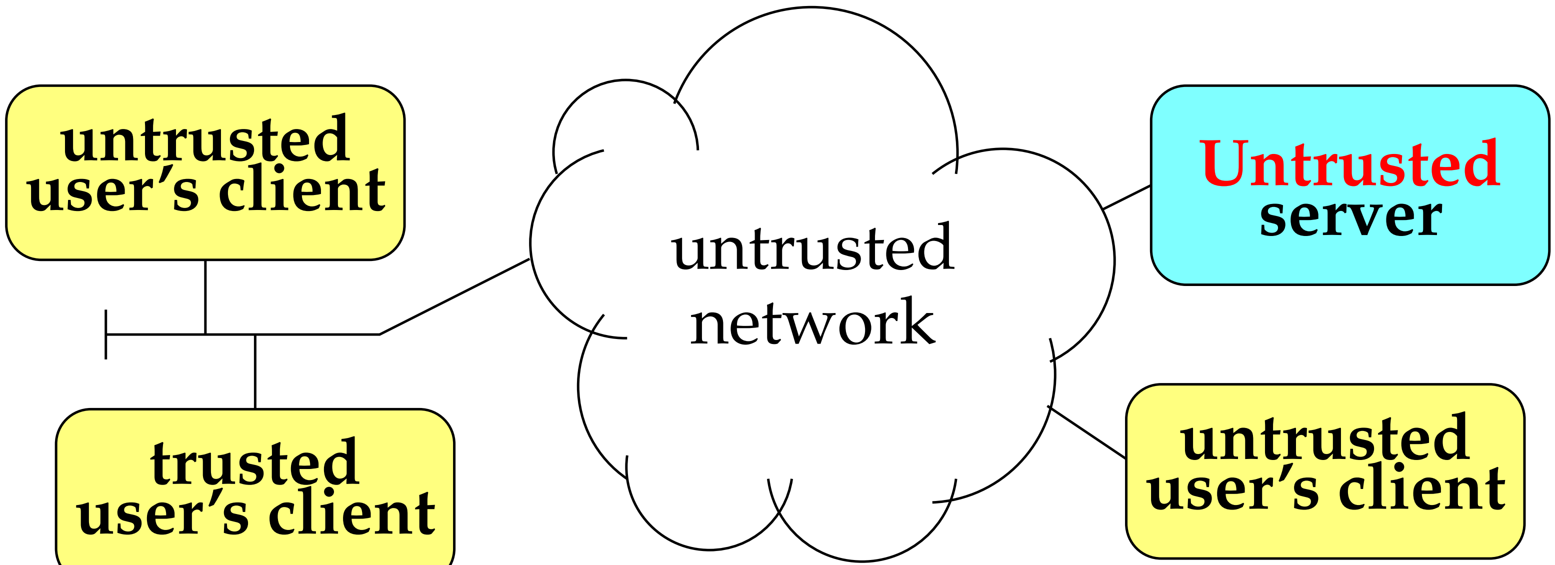
Definition. A set $\mathcal{O}$ of fetch & modify operations is **fetch-modify consistent** iff $\mathcal{O}$ is orderable and any fetch F of a file p reflects exactly the modifications of p that happened before F.

Traditional secure network file systems



- **Users are untrusted and control their own clients**
- **Server trusted to reflect only authorized modifications**
- **All communications mutually authenticated**
  - Server knows which user (principal) issued each request
  - Clients know responses come from server

The SUNDR file service



- **Eliminates trust in server**
  - Users certify data when they store it to server
  - Clients can verify data without trusting the server
  - E.g., Must penetrate trusted user's client to compromise FS
- **Any server misbehavior easily detectable**

Related work: Cryptographic storage

- **Old idea: 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

SUNDR approach

- **Assume digital signatures much cheaper than net. RTT**
  - Increasingly valid assumption as CPUs improve
- **Give server + every user a public signature key**
  - Assume all parties know the others' keys
    (Can actually 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 each others' signed data for consistency
- **Any server misbehavior then readily detectable**

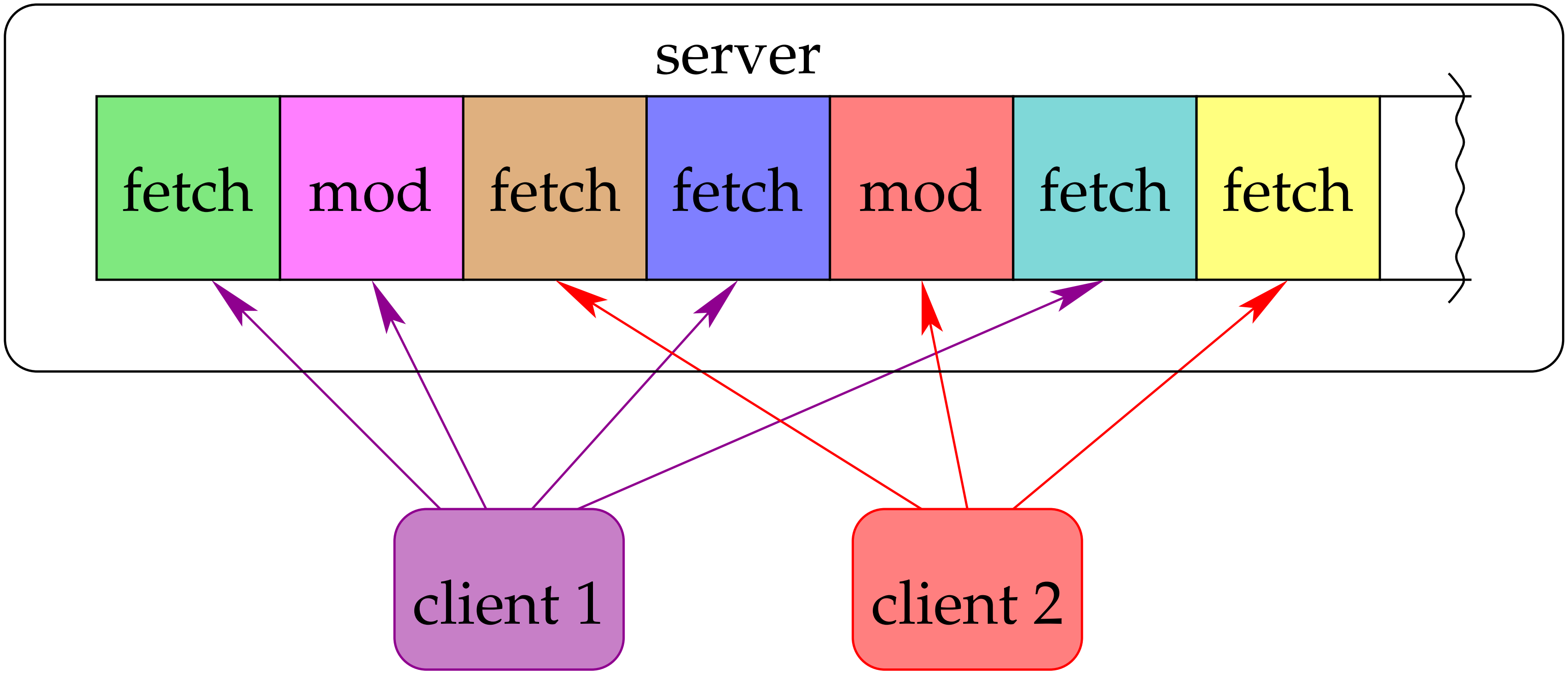
The consistency problem

- **W/o on-line trusted party, consistency complicated**
  - No way for two parties to communicate reliably if never both on-line simultaneously
  - Yet users are the trusted entities, and not always on-line
- **Consider the following failure (attack) of server:**
  - User A logs in, modifies a file, logs out
  - User B doesn't know if A logged in or not
  - Malicious server hides A's changes from B (undetectable)

Limits of untrusted servers

- **Cannot guarantee fetch-modify consistency**
- **Yet want consistency failures to be detected**
- **What can one do with untrusted servers?**
- **Idea: Any consistency failure should cause all hell to break loose**
  - Magnify subtle failures to readily detectable ones
  - Communicating clients can then audit server
  - Even humans will likely notice problem in conversation

Straw man implementation: Signed history



- **Server keeps total history of all operations**
- **Each element contains signature of past history**
  - No concurrent operations (server provides untrusted lock)
  - Clients check each other's signatures to verify file contents

Consistency semantics

- **Clients must agree on complete history of FS**
  - Check any two histories by seeing if one is prefix of other

fetch	mod	fetch	fetch	mod	fetch	fetch

- **Consistency violations produce incompatible histories:**

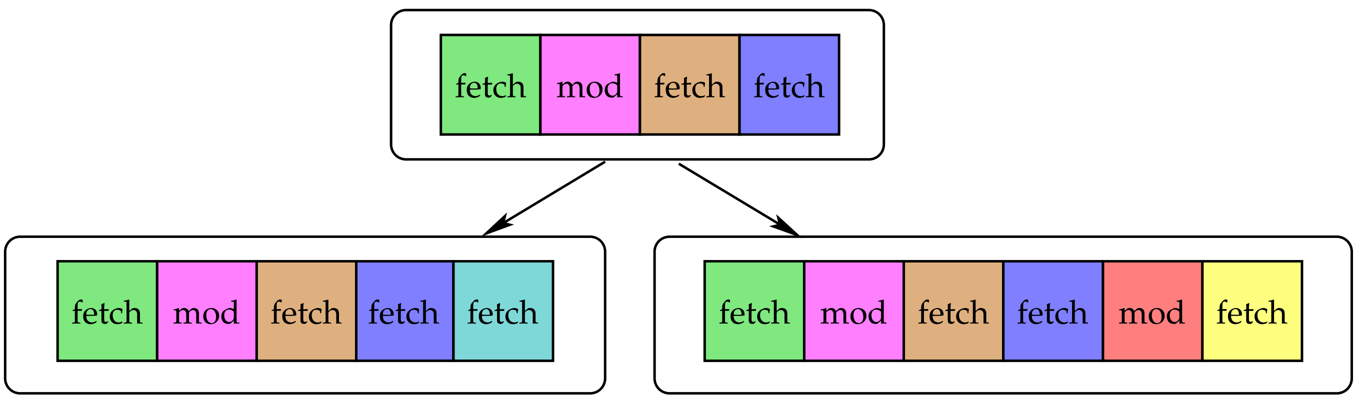
fetch	mod	fetch	fetch	fetch	(sig by client 1)

fetch	mod	fetch	fetch	mod	fetch	(sig by client 2)

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

Forking tree

- **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 as a graph, edge to node from 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**

Why fork consistency?

- **We needed a relaxed notion of consistency**
- **Fork consistency magnifies subtle 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
- **Users can trivially audit server retroactively**
  - If you see effects of operation X, guarantees file system was consistent at least until X was performed
  - Exchange information about a recently modified file
  - Clients that communicate get fetch-modify consistency
  - Pre-arrange for "timestamp" box to update FS once per day

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
- If g' is parent of g, $\forall O_{\in g}(O \in g'$ or $\forall O'_{\in g'}\ O' \prec O)$

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

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.

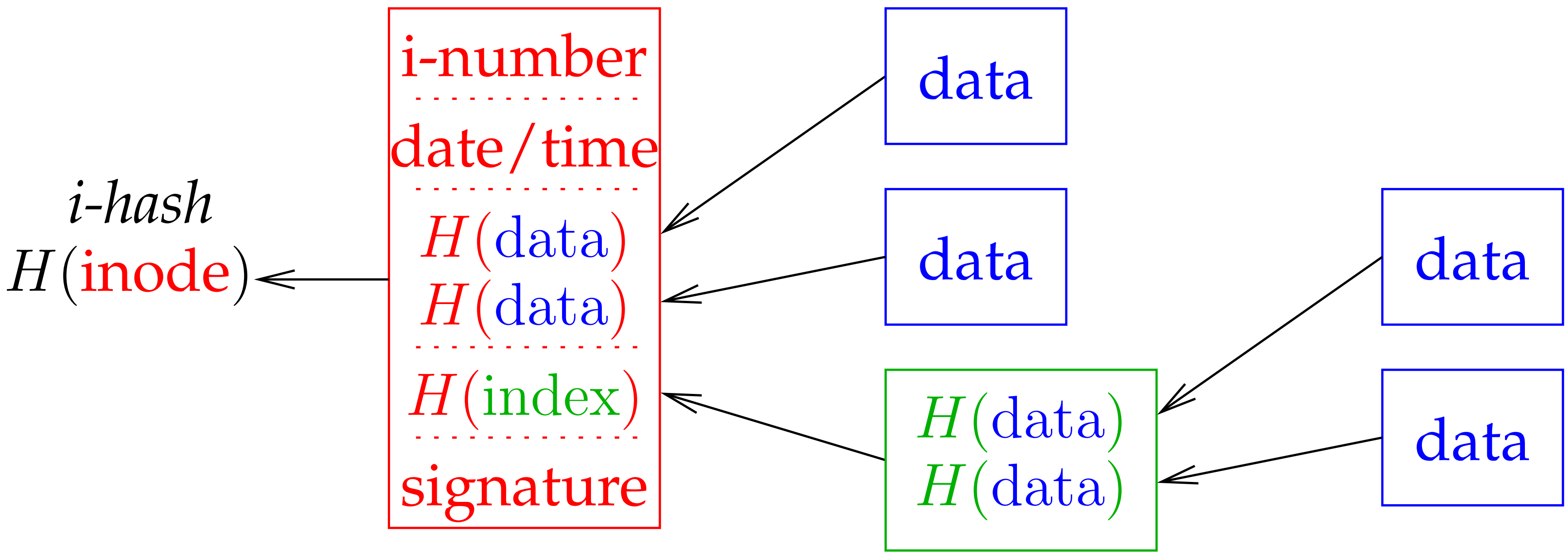
Proof (sketch): Consider set $\{o \mid o \leq q\}$ for each maximal operation, & longest prefixes, as with history.

Implementing fork consistency

- **Signing complete histories not practical**
- **SUNDR takes a more efficient, two-pronged approach**
  - All files that each user or group can write are certified with a short *i-handle*
  - Special protocol for fetch/mod of i-handles
- **Relies heavily on collision-resistant hash functions (Computationally infeasible to find $x \neq y,\ H(x) = H(y)$)**

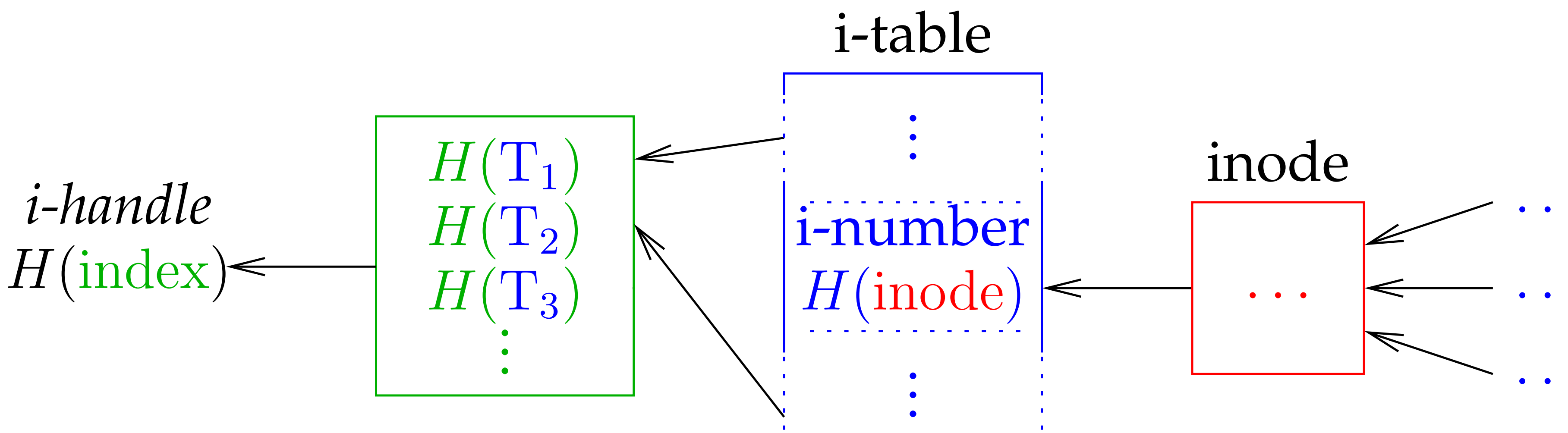
Certifying files

- Goal: Short value from which file can be verified
- Use recursive hashing for efficient random access



- Given i-hash, can verify any block of file
- Problem: Must interpreted i-hashes in context

Digitally signing file systems



- **Recursively hash FS data structures [SFSRO]**
  - inode specifies file contents,
  - i-table specifies i-number $\rightarrow$ inode bindings,
  - i-handle specifies i-table, and thus user's file data
- **Each user digitally signs his own i-handle**
  - Directories map filename $\rightarrow$ $\langle$user, i-number$\rangle$

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 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

Implementing i-handle consistency

- **User assigns increasing vers. no. to their i-handle**
- **Idea: Users sign each other's version numbers:**
  - Each user u_i maintains a *version structure*:
    $y = \{\text{VRS}, \text{i-handle}, u_1\text{-}n_1\ u_2\text{-}n_2\ \ldots\ u_i\text{-}n_i\ \ldots\}$
  - When updating his i-handle, a user bumps his own version
    $\{\text{VRS}, u_i\text{-}h, u_1\text{-}n_1\ u_2\text{-}n_2\ \ldots\ u_i\text{-}(n_i+1)\ \ldots\}_{K_{u_i}^{-1}}$
  - When updating a group, a user bumps his & group's no.:
    $\{\text{VRS}, u_i\text{-}h\ g\text{-}h_g, u_1\text{-}n_1\ u_2\text{-}n_2\ \ldots\ g\text{-}(n_g+1)\ \ldots\ u_i\text{-}(n_i+1)\ \ldots\}_{K_{u_i}^{-1}}$
- **All signed version structures must be ordered**
  - Let $y[u]$ by 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, user u_i’s client:**
  - UPDATE: Locks FS, downloads and sanity checks VSL
  - Calculates & signs new version structure:
    $\{\mathrm{VRS}, u_i\text{-}h, u_1\text{-}n_1\ u_2\text{-}n_2\ \ldots\ u_i\text{-}n_i\ \ldots\}_{K_{u_i}^{-1}}$
  - COMMIT: Uploads version struct for new VSL, releases lock

Example

Users u and v both start at version 1:

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

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

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

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

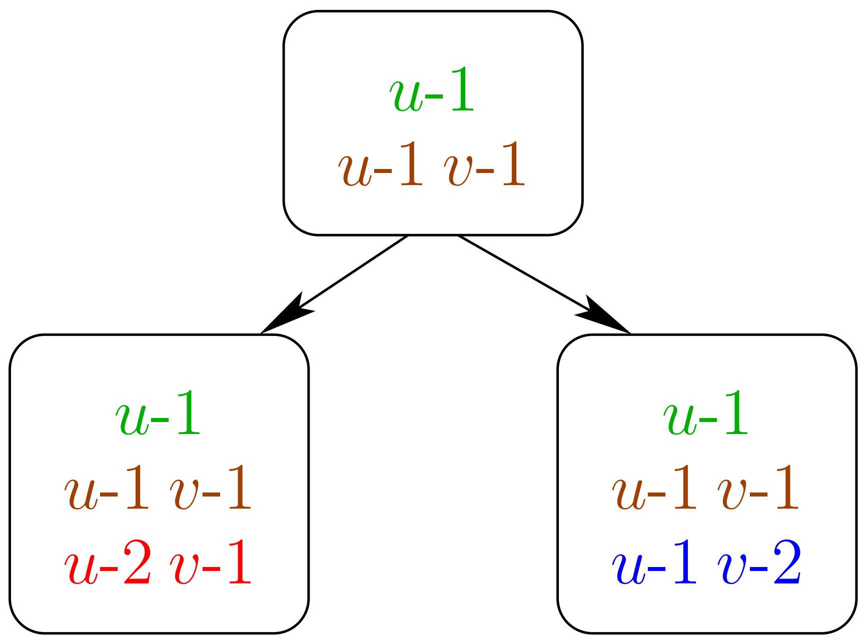
$$
\begin{aligned}
y_u &= \{\mathrm{VRS}, u\text{-}h'_u, u\text{-}2\ v\text{-}1\ \ldots\}_{K_u^{-1}} \\
y_v &= \{\mathrm{VRS}, v\text{-}h_v, u\text{-}2\ v\text{-}2\ \ldots\}_{K_v^{-1}}
\end{aligned}
$$

Attack

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

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

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



Concurrent updates

- **Bad to lock FS between UPDATE & COMMIT**
- **Fix: pre-declare operations in UPDATE certificate**
  $\{\textbf{UPDATE}, u, n+1, H(y_u), [\langle \textbf{usr/grp}, \textbf{inum}, \textbf{ihash}\rangle, \ldots]\}_{K_{u_i}^{-1}}$
  - Specify new version number, hash of old version struct
  - Specify new i-hashes for any modified files (deltas for dirs)
- **Server keeps list of pending updates in *pending version list* or PVL**
  - Replies to UPDATE by sending 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

Concurrent protocol details

- **Version structures now reflect pending updates**
  - 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 $\perp$: by convention, u's nth version struct 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 $x \le y$ iff:

1. For all users u, $x[u] \le y[u]$ (i.e., $x \le y$ by old def)
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 improvements

- **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)**

Conclusions

- **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