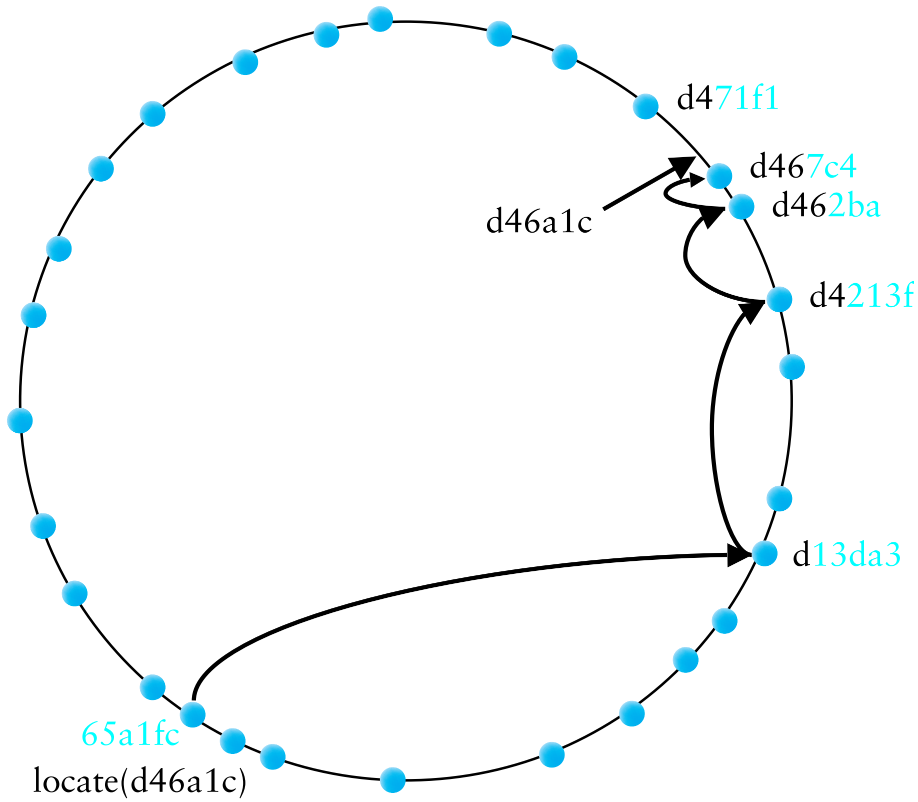
Review: Pastry routing tables

Row 0	*0x*	*1x*	*2x*	*3x*	*4x*	*5x*		*7x*	*8x*	*9x*	*ax*	*bx*	*cx*	*dx*	*ex*	*fx*
Row 1	*60x*	*61x*	*62x*	*63x*	*64x*		*66x*	*67x*	*68x*	*69x*	*6ax*	*6bx*	*6cx*	*6dx*	*6ex*	*6fx*
Row 2	*650x*	*651x*	*652x*	*653x*	*654x*	*655x*	*656x*	*657x*	*658x*	*659x*		*65bx*	*65cx*	*65dx*	*65ex*	*65fx*
Row 3	*65a0x*		*65a2x*	*65a3x*	*65a4x*	*65a5x*	*65a6x*	*65a7x*	*65a8x*	*65a9x*	*65aax*	*65abx*	*65acx*	*65adx*	*65aex*	*65afx*

- **Routing table of node with ID i =65a1fc$x$'s**
  - For each prefix p of i, and each digit $d \in [0 \ldots \mathrm{f}]$, has contact with ID prefix pd

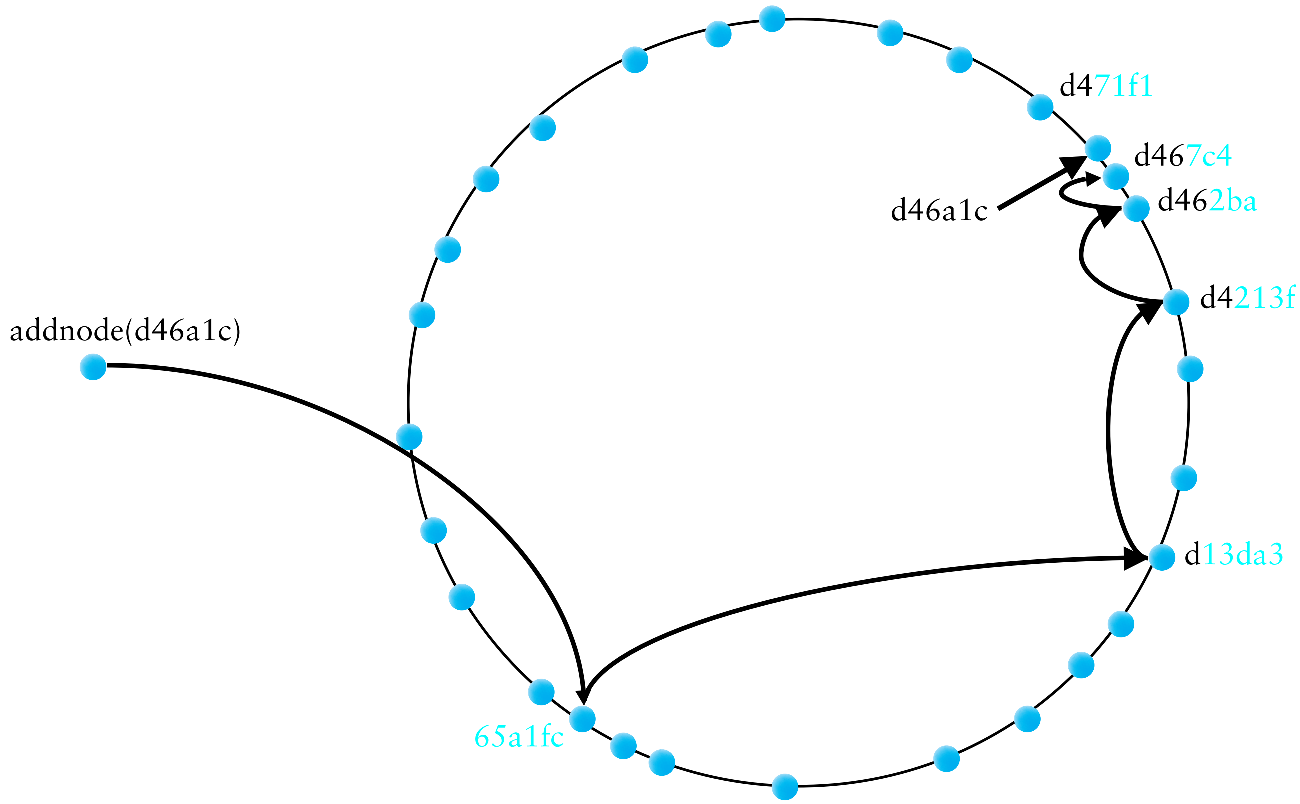
Looking up objects in Pastry



- **"Corrects" one digit at a time**
  - Queries nodes with IDs: d13da3, d4213f, d462ba
  - Then use "leaf set" to find node with nearest ID to target

Joining the system

- **Must know of one existing node in system**
  - Query it and other nodes to find node closest to your ID



- **Initialize leaf table from node closes to ID**
  - Will know almost complete leaf set for new node

Initializing routing table

- **Can't initialize routing table from closest node**
  - E.g., 1fffff$x$'s closest node might be 200000y
- **But can fill up routing table from intermediate nodes**
  - Can use entire first row of first node contacted
  - Use second row of second node contacted, since same first digit as joining node
- **Once join procedure complete, can issue queries**
  - New node knows enough to route to and ID
- **But what about queries *to* IDs near new node?**

Fixing up state for a join

- **New node must fix *other* nodes' routing tables as well as initialize its own**
- **For correctness, need to fix up neighbor's leaf sets**
  - Easy, node can contact them after initializing its own leaf set
  - If leaf sets correct, routing works, but could take $O(N)$ hops
- **Updating other nodes' routing tables:**
  - Old routing tables either correct, or missing entry new node could fill
  - Automatically fill holes as side affect of lookups
  - New node sends its state to each node in its routing table
  - Nodes periodically query to try to fill holes in their tables

Node failure

- **Nodes can fail without warning**
  - Other node's routing tables & leafs sets point to dead node
- **Routing table: Detect timeout, treat as empty slot**
  - Route to numerically closest available
  - Repair: Ask any node on same row for its contact
    Or ask any node below, since all will have correct prefix
- **Leaf sets: Node closest to target could be dead**
  - Need to find next closest
  - That's why leaf sets not just one neighbor ($O(\log N)$)
  - Easy to update leaf sets by contacting other nearby nodes

How reliable is Pastry?

- **For correctness, only need leaf sets**
- **Assume independent node failures**
  - Each node fails with probability p in maintenance interval
  - Say leaf set contains L values
  - Probability of being cut off is p^L
  - So for large N, if $L \sim \log N$, pretty good
- **Is independent failure a reasonable assumption?**
  - Good that $\mathrm{nodeID} = \mathrm{MD5(IP\ Address)}$
  - Proximity in ID space not correlated with physical proximity
  - But big network outages, synchronized renumbering correlated

Locality

- **Lookup takes $O(\log n)$ hops**
  - But hops could be long (NYC→Germany→LA)
- **Note: Many options for top levels of routing table**
  - Can chose *any* node with correct prefix
  - So pick nodes that are close to you to speed lookup
  - But makes it harder to assume independent failures
- **Continuously adjust routing table for locality**
  - Asks current entry for that entry's complete tables
  - Ping suitable nodes from other node's tables
  - Use them instead of current entry if ping says closer
- **No choice for leaf sets, though**

Short routes property

- **Locality optimization helps recursive lookups**
  - New node will know of nodes close to it
  - Very good if triangle property holds (X close to Y and Y close to Z $\Longrightarrow$ X close to Z)
  - Often does hold, but not always
- **This is known as *short routes property***
  - Individual hops are lower latency
  - But less and less choice (lower node density) as you get close in ID space
  - So last few hops likely to be very long.
  - You don't *end up* close to the initiating node, just get there more quickly

Scribe

- **Pastry can be used to form multicast trees**
  - Hash name of multicast group to get ID
  - Node closest to ID is *rendez-vous point* or root
  - To multicast a message, deliver it do RP, which sends it down the tree
- **Form multicast tree by routing JOIN msgs to ID**
  - Each node keeps track of groups + children for each group
  - On receipt of JOIN message, add sender to children
  - If child joins a new group, send join to parent
    (parent is just next hop towards ID)
- **Send just proceeds from RP to leaves**
  - Senders cache IP address of RP to save upwards routing
- **Leave protocol similar to join**

Scribe locality

- **Short routes property helps multicast trees**
  - Towards leaves, parents are in high-levels of routing table
  - These are precisely the contacts with best locality
  - So often delivering messages to nearby nodes
  - Which may well reduce link stress (e.g., node 1abc$x$ at NYU will chose node 2def$y$ at NYU over farther nodes)
- **"Bottleneck remover" algorithm for overload**
  - Node may decide it is forwarding too many copies
  - Measures children & boots furthest away
  - Booted node effectively gets pushed down the tree

Reliability & failure

- **Scribe sends messages over TCP**
  - But doesn’t guarantee reliability
  - Nodes can crash and leave system abruptly
  - In fact, Sbribe itself doesn’t guarantee reliable delivery
- **Detect failures using heartbeat messages**
  - Each non-leaf node periodically sends heartbeat to children
  - Any multicast message serves as a heartbeat
  - So only need extra traffic when group quiescent
  - Upon timeout, route around failed node in Pastry
- **Must replicate root state in case root fails**
  - Typically replicated on 5 nearest nodes to ID

Reliable/ordered multicast

- **Can build reliable/ordered multicast on Scribe**
- **Source assigns sequence number to each message**
  - Nodes do not send messages out of order
  - Wait for all previous messages before sending next
  - After fault+repair, you know what you are missing
- **Nodes buffer old messages**
  - Keep around for longer than detect+repair time
  - So when you repair, can request messages you missed

Splitstream

- **Problem: Scribe makes uneven use of resources**
- **In fully-balanced tree w. height h, fanout f**
  - f^h leaf nodes consume no upstream b/w
  - $(f^h - 1)/(f - 1)$ internal nodes consume $f\times$ stream b/w
  - E.g., with $f = 16$, $< 10\%$ of nodes carry forwarding load!
- **Better approach: Stripe data over a *forest* of trees**
  - Each node is leaf in some, internal in others
  - Could round-robin packets down multicast trees
  - Or could stripe at the bit level
  - One tree could be parity bit, to survive a failure

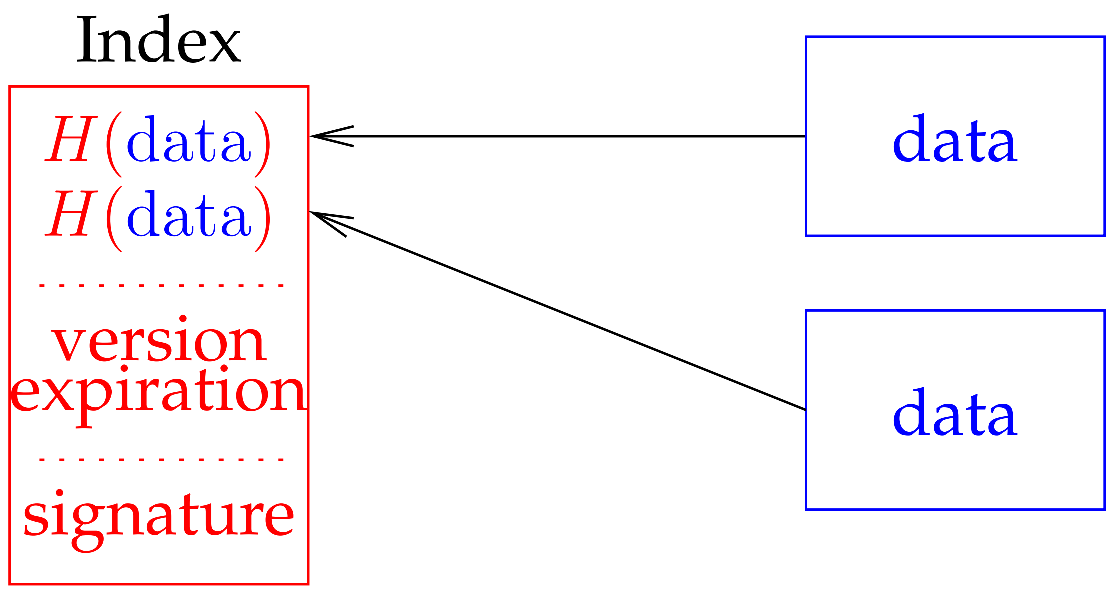
Interior-node-disjoint trees

- **Want to avoid a failure affecting multiple streams**
  - E.g., say node n is your ancestor in multiple trees
  - If n fails, you lose multiple streams (so parity won't help)
- **Solution: Each node is interior in only one tree**
  - Say digits are in base 16
  - Can achieve by having 16 trees, each with a group ID that starts with a different digit
  - Can only be interior node if group ID and you have at least one-digit prefix in common

CFS

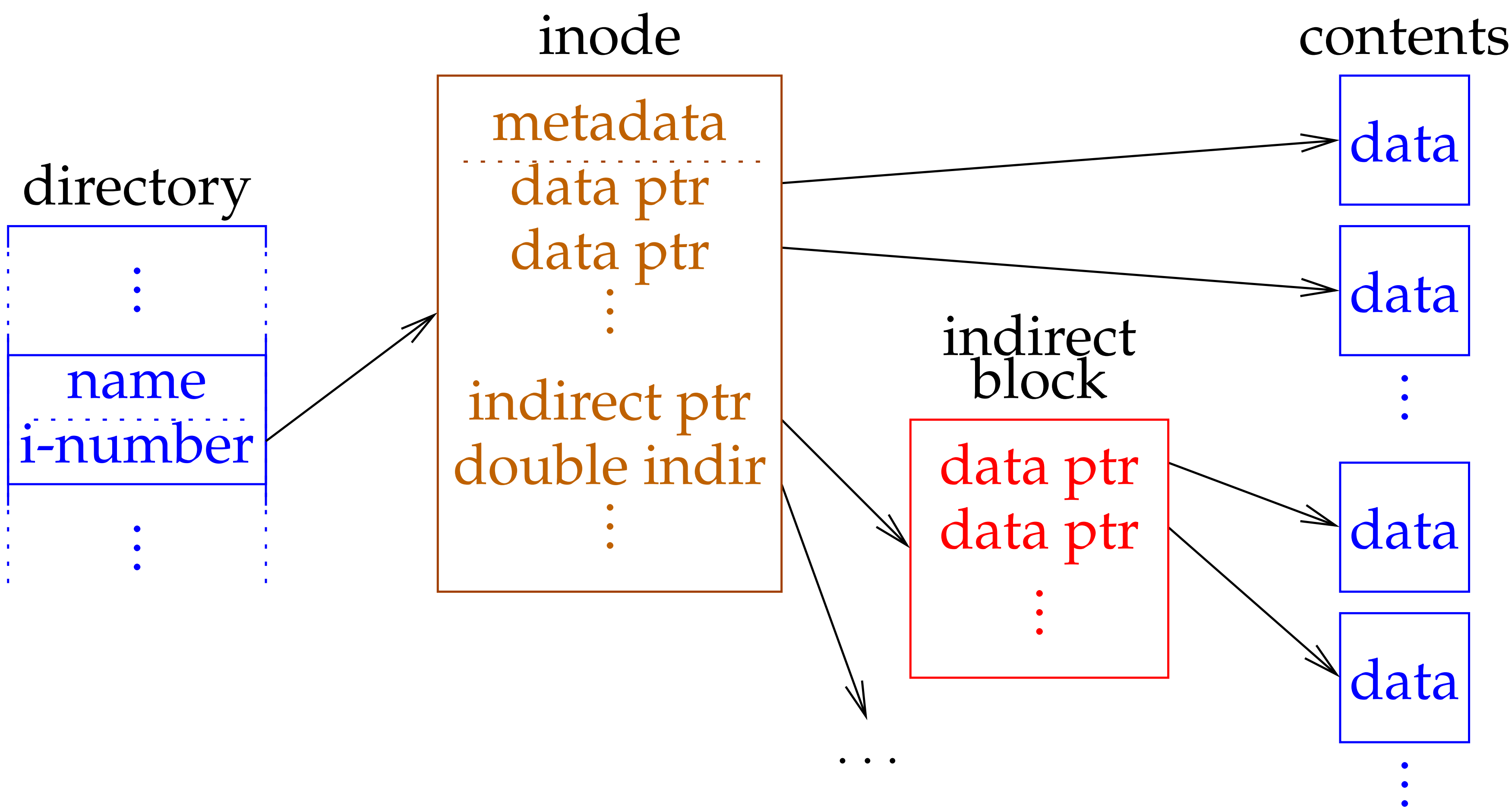
- **Another application of P2P systems**
- **Idea: Replicate widely stored data in DHT**
  - E.g., Linux distribution
  - Care a lot about data integrity—no tampering!
- **CFS – cooperative file system is P2P file system**
  - Read-mostly file system
  - Publish operation breaks into blocks
  - Spreads chunks all around DHT
  - Digitally sign entire file system for integrity

Example: Publishing 2 blocks of data



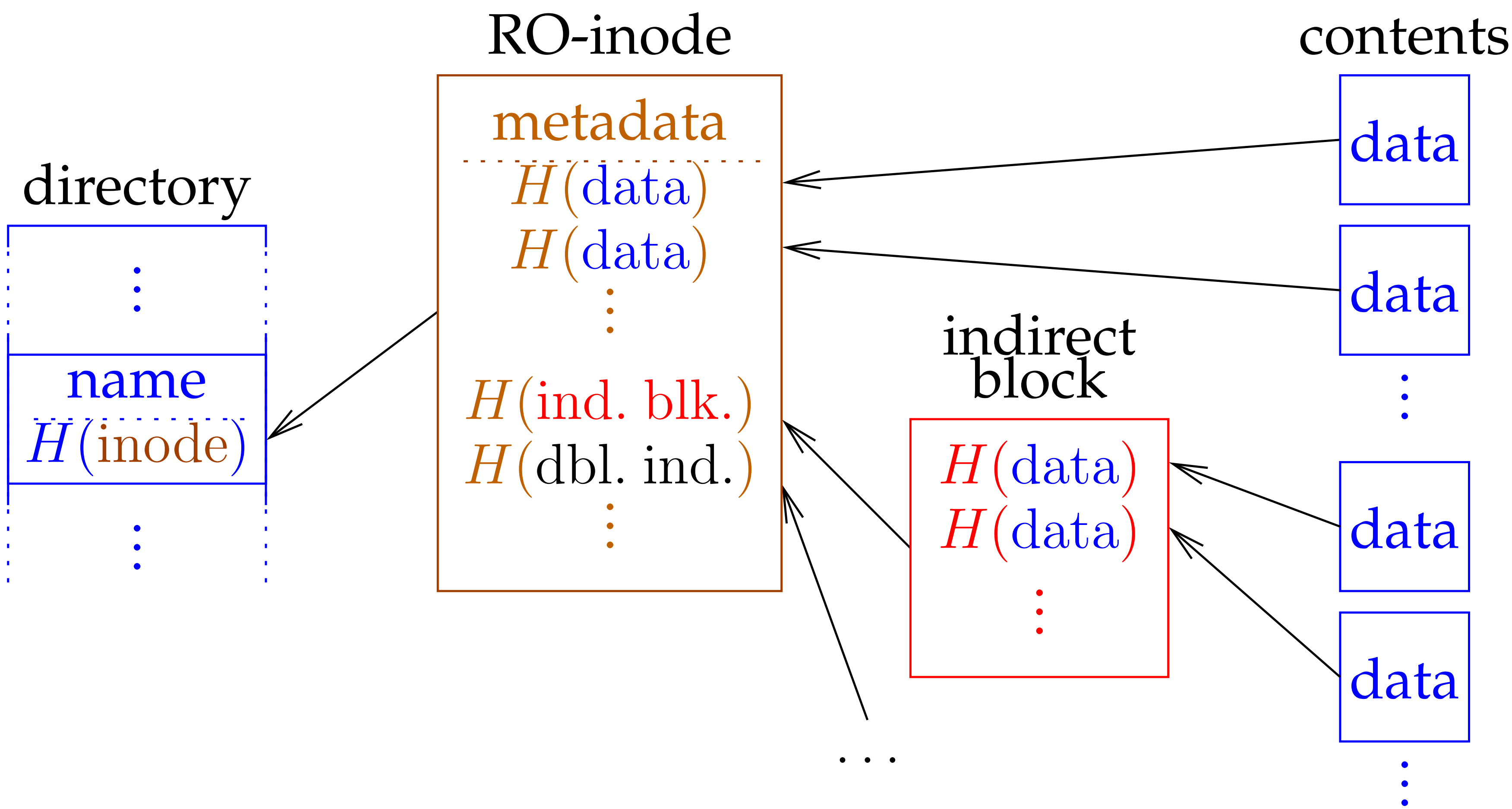
- **Digitally sign version & hashes of blocks**
  - Can verify one block without having the other
  - Two blocks must come from same version of file
- **Generalize technique to an entire file system**

Traditional FS data structures



- **In database arbitrary key can replace disk location**

CFS data structures



- **Index all data & metadata with cryptographic hash**

CFS scalability & reliability

- **CFS built on Chord not Pastry, but ideas similar**
- **Blocks must be replicated for reliability**
  - Easy: Store each item at k successor nodes around circle
- **Blocks must be replicated for scalability**
  - E.g., Imagine everybody reads the same block
  - Don't want to overload poor successor node
- **Solution: Store blocks along the lookup path**
  - Suppose you are looking up block B on node n_0
  - You may traverse nodes n_3, n_2, n_1, n_0 to get B
  - Now store B on n_1
  - Next lookup that converges at n_1 will store on prev, etc.