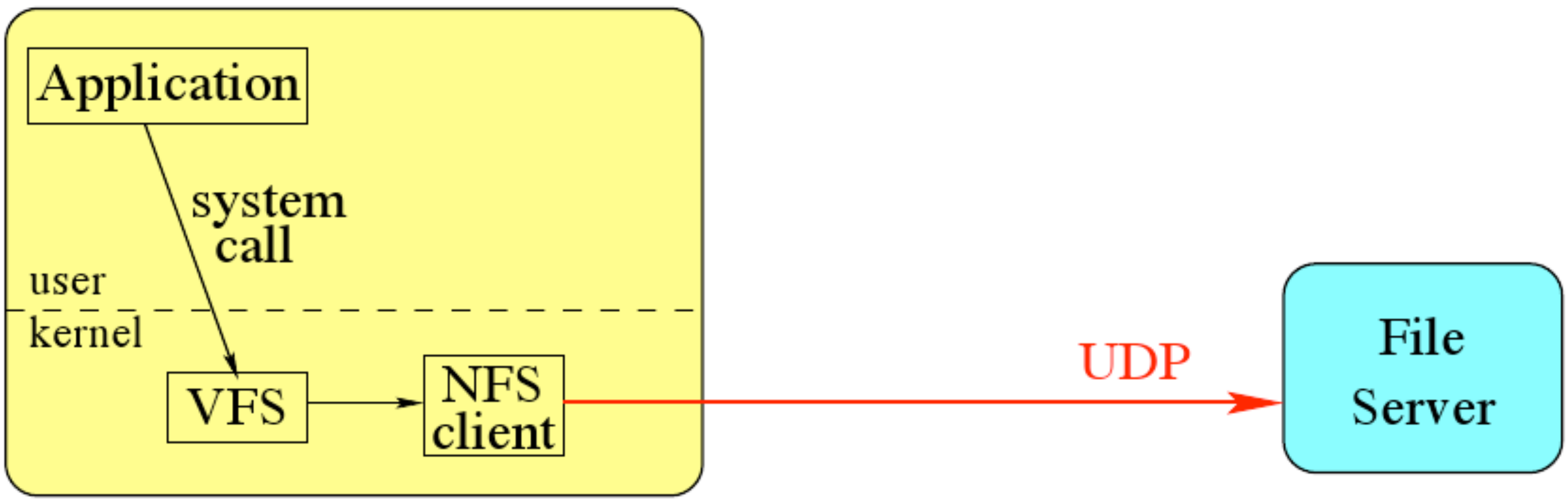
Cooperative File System

So far we had...

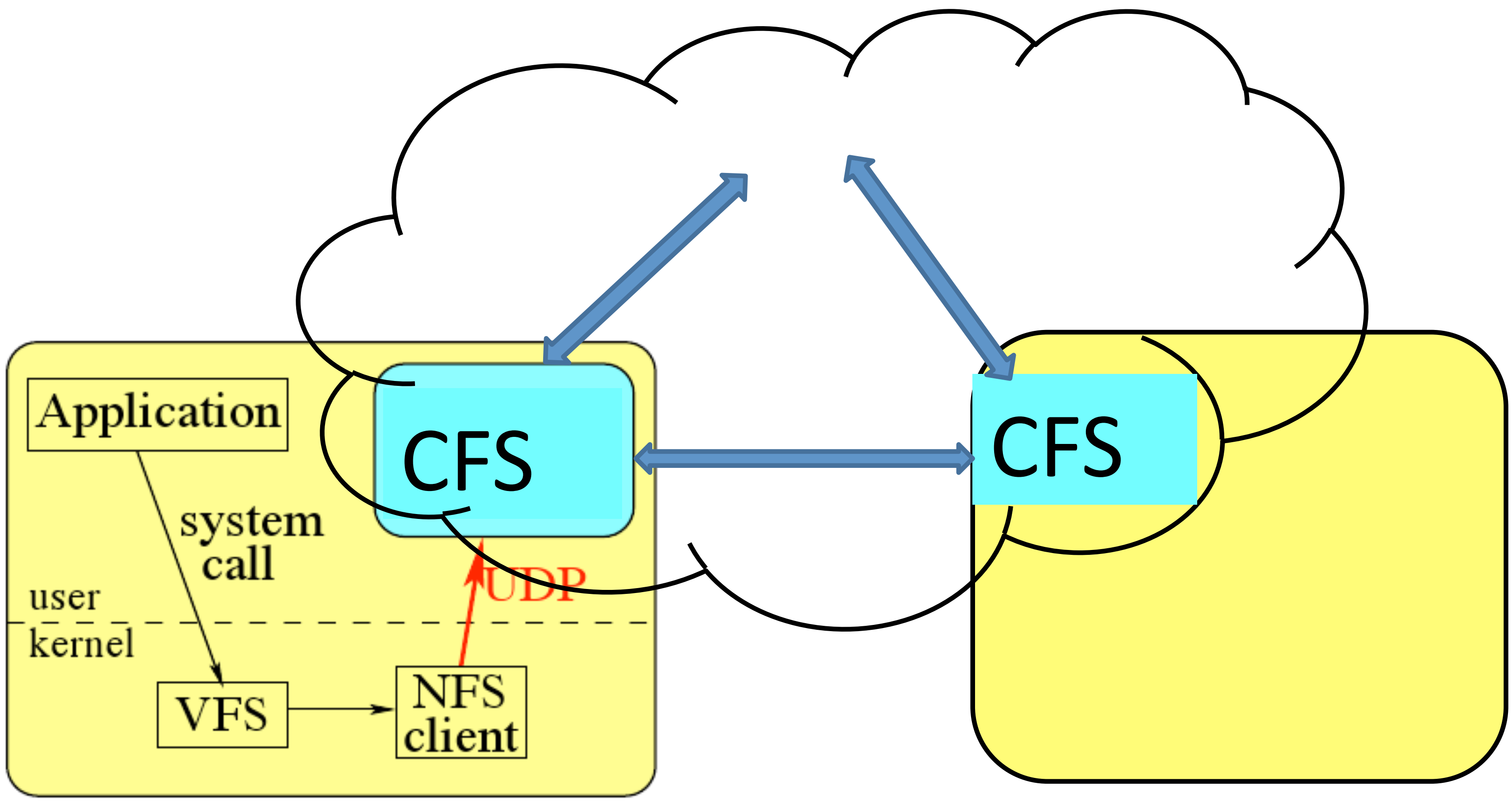


- Consistency BUT...

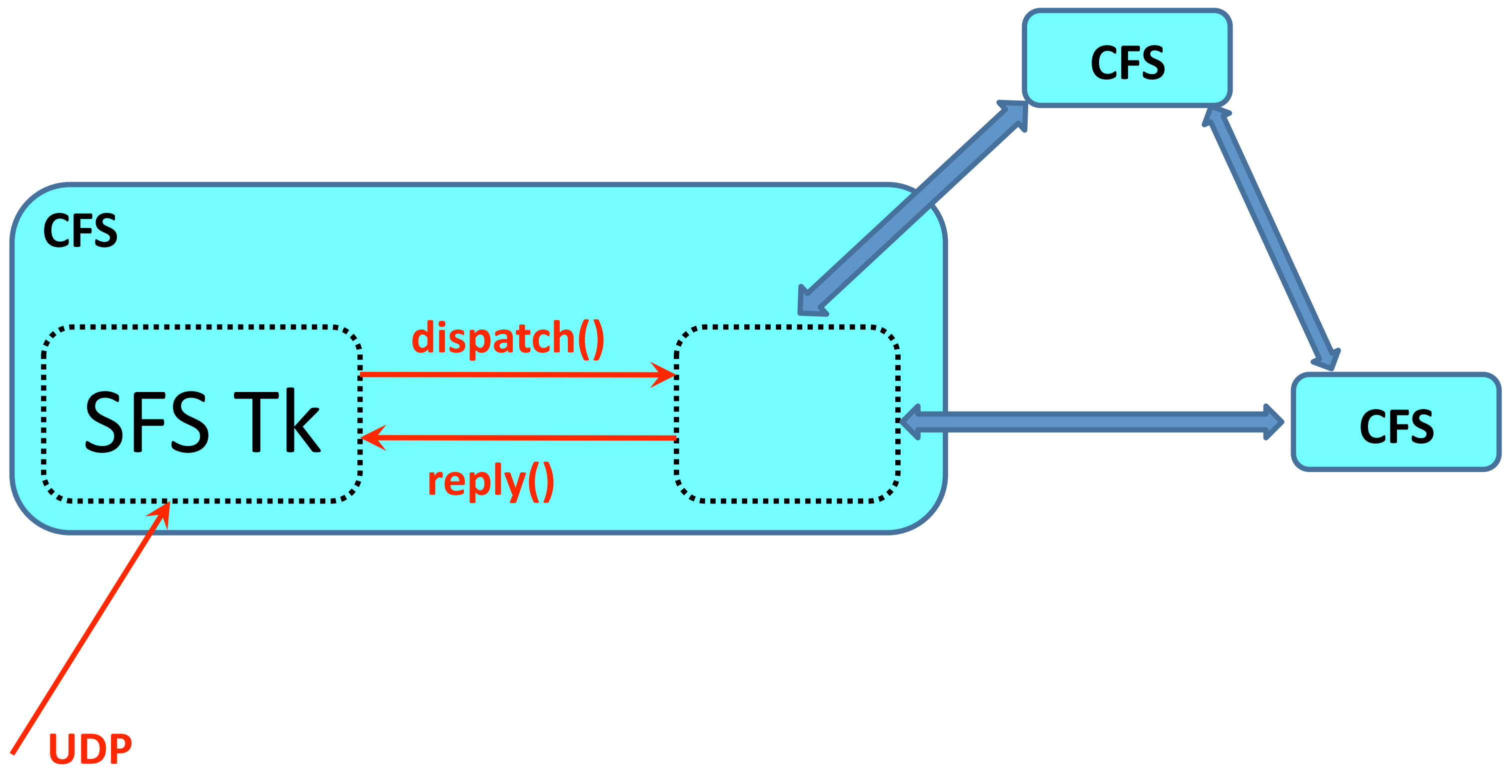
- Availability

- Partition tolerance ?

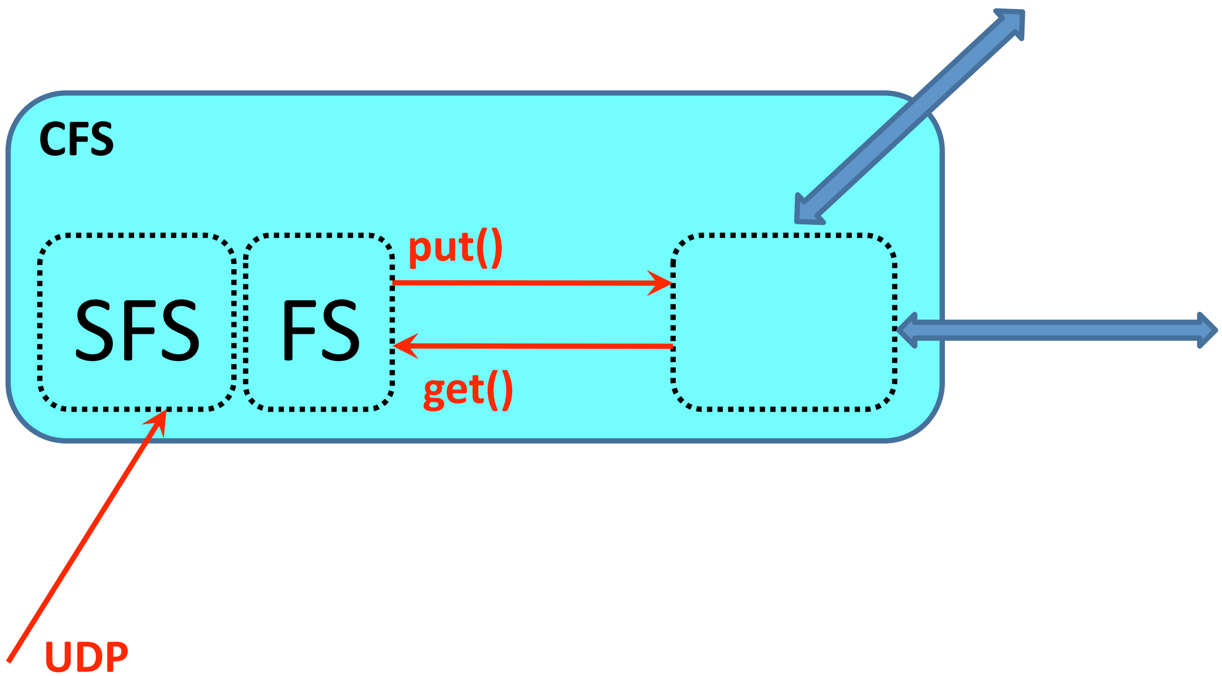
Let's be cooperative



We need a NFS Server…
…let's use the SFS toolkit



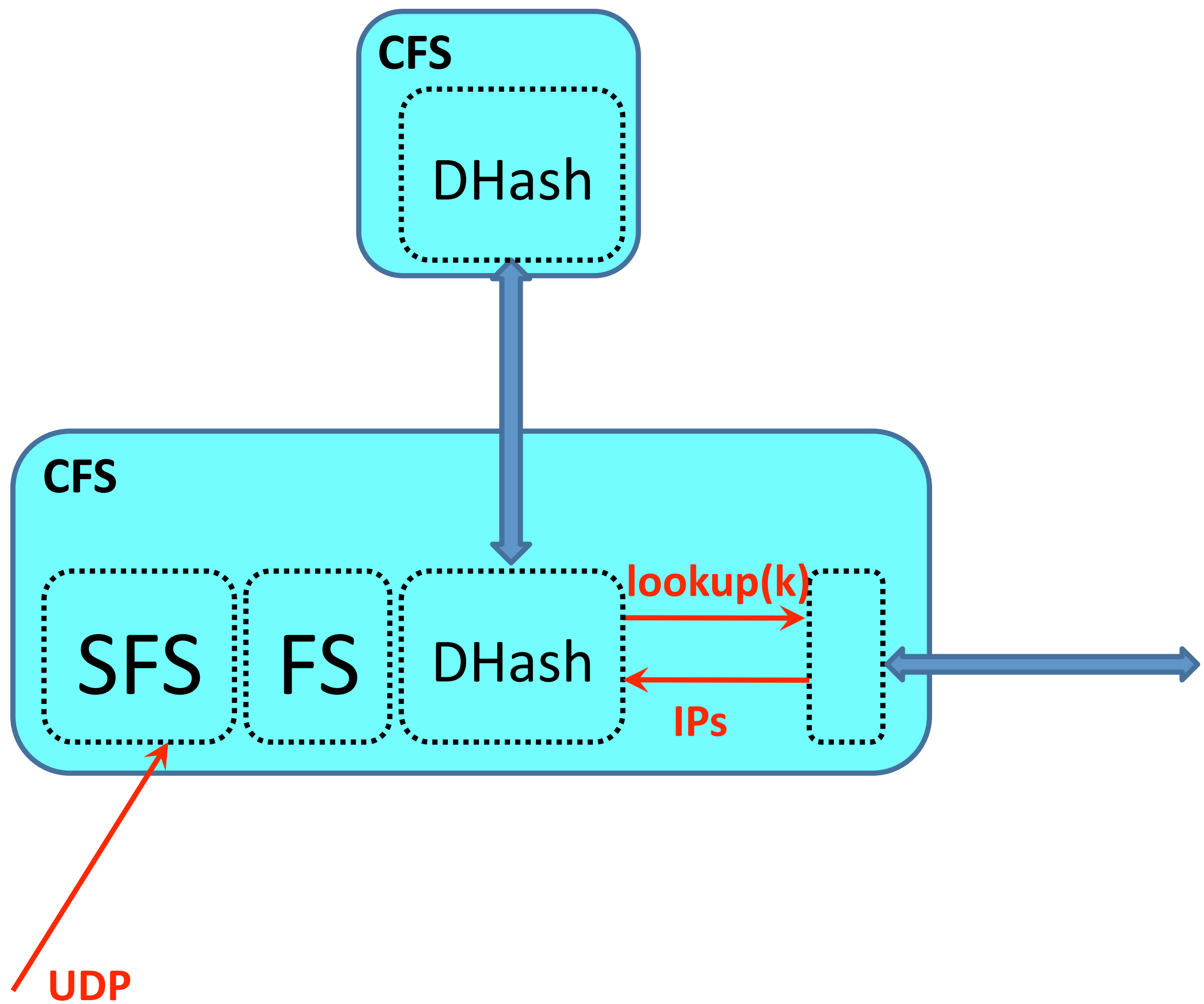
Small is beautiful



- Need to emulate a real FS
- Small entities to spread the load

- Let's implement a real FS,
- where blocks are spread over multiple peers,
- and identified by keys

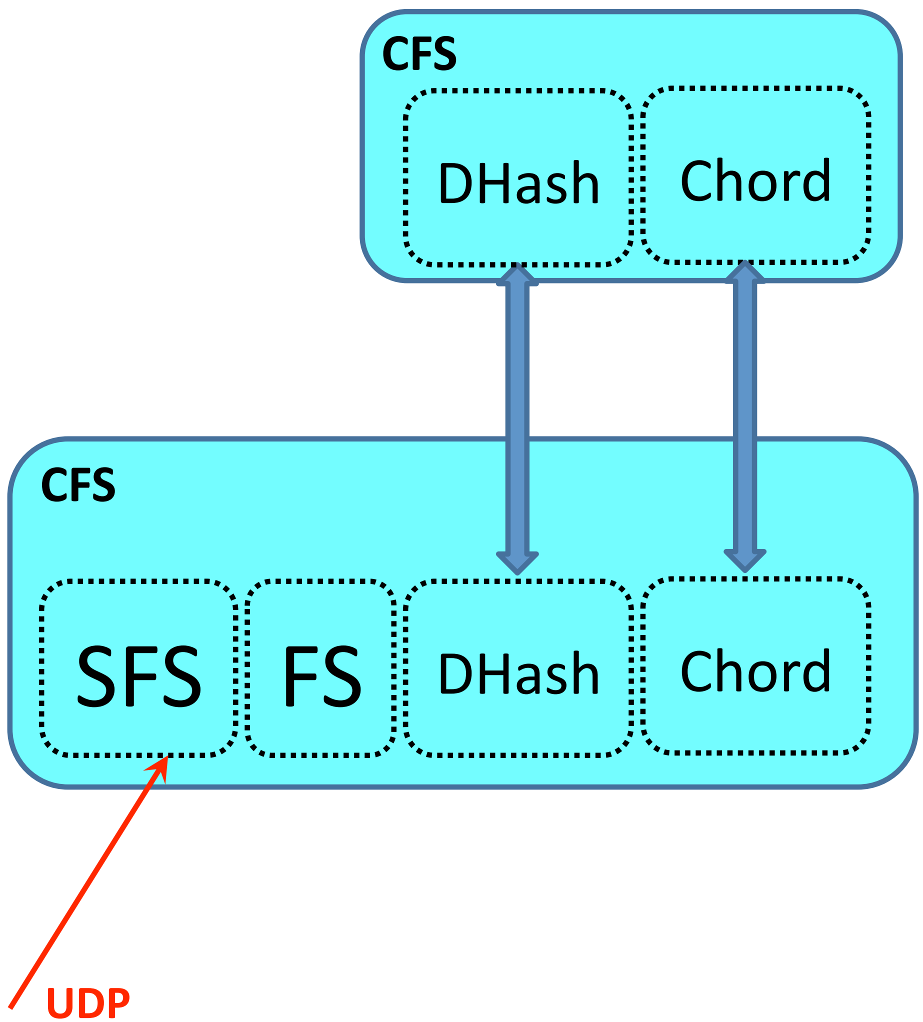
Replication is everything



- Servers fails
- Servers may be overloaded

- Let's replicate
- We need a layer to locate them

Who has the key?



- We need to associate each key to an IP,
- in a distributed manner,
- with efficient lookup
- Let's use Chord

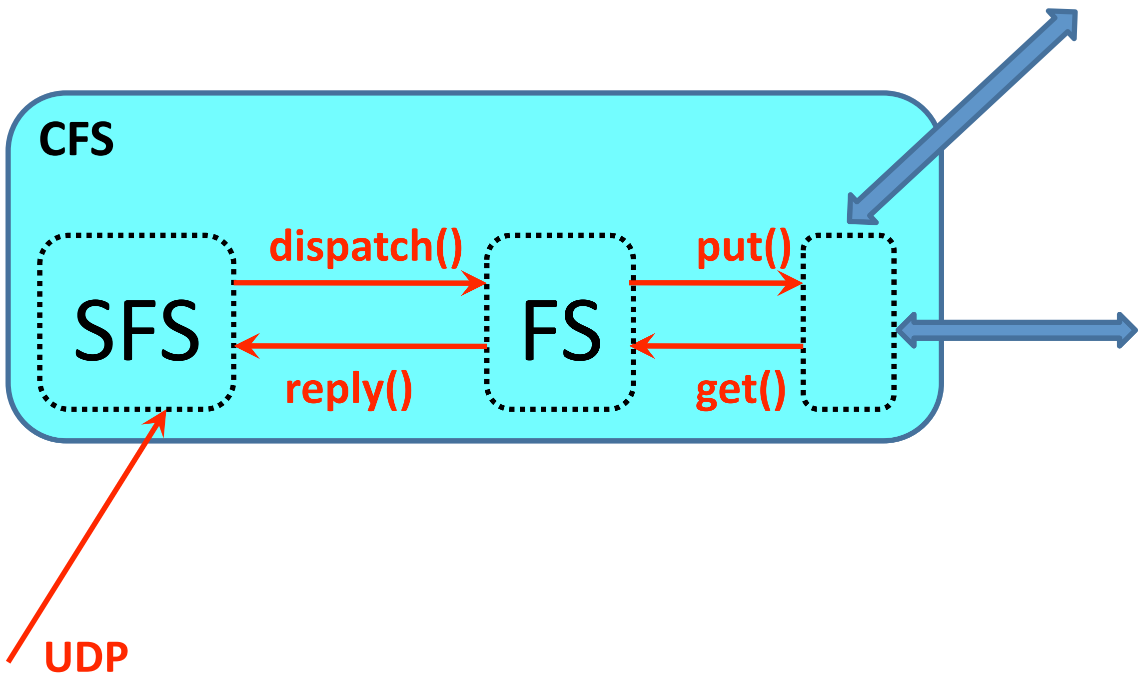
Outline

- Overview
- Objectives
- FS
- DHash
- Chord
- Discussion

Objectives

- Decentralized control
- Scalability
- Availability
- Load Balance
- Persistence
- Quotas
- Efficiency
- Consistency ?

The File System

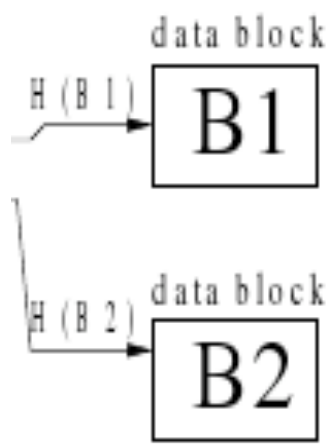


- Convert NFS calls into put and get primitives
- Integrity
- Consistency

Let’s store a block

- How do we enforce integrity?
- Server can be malicious…
- Everyone can write a block at any given key…
- We can either:
  - Sign the block
  - Store at its hash value
- We want load balancing => K=H(B)
- Is it enough?
- Billion users X Billion block per user => 10^{-28}

Let’s store a File and a Directory

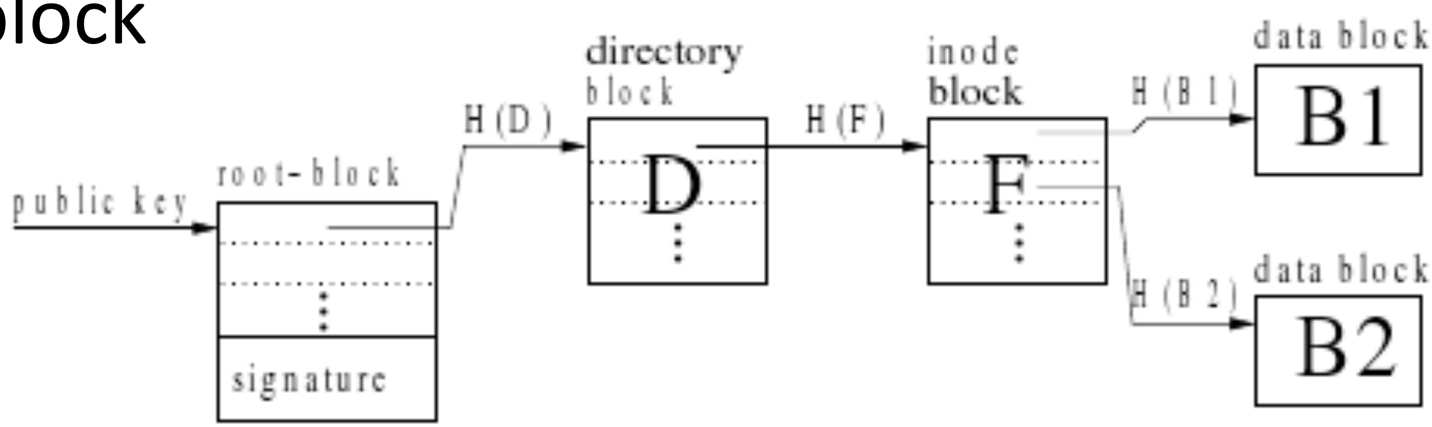


- Remember the blocks’ keys
- Use an inode block
- => File Key = H(inode)

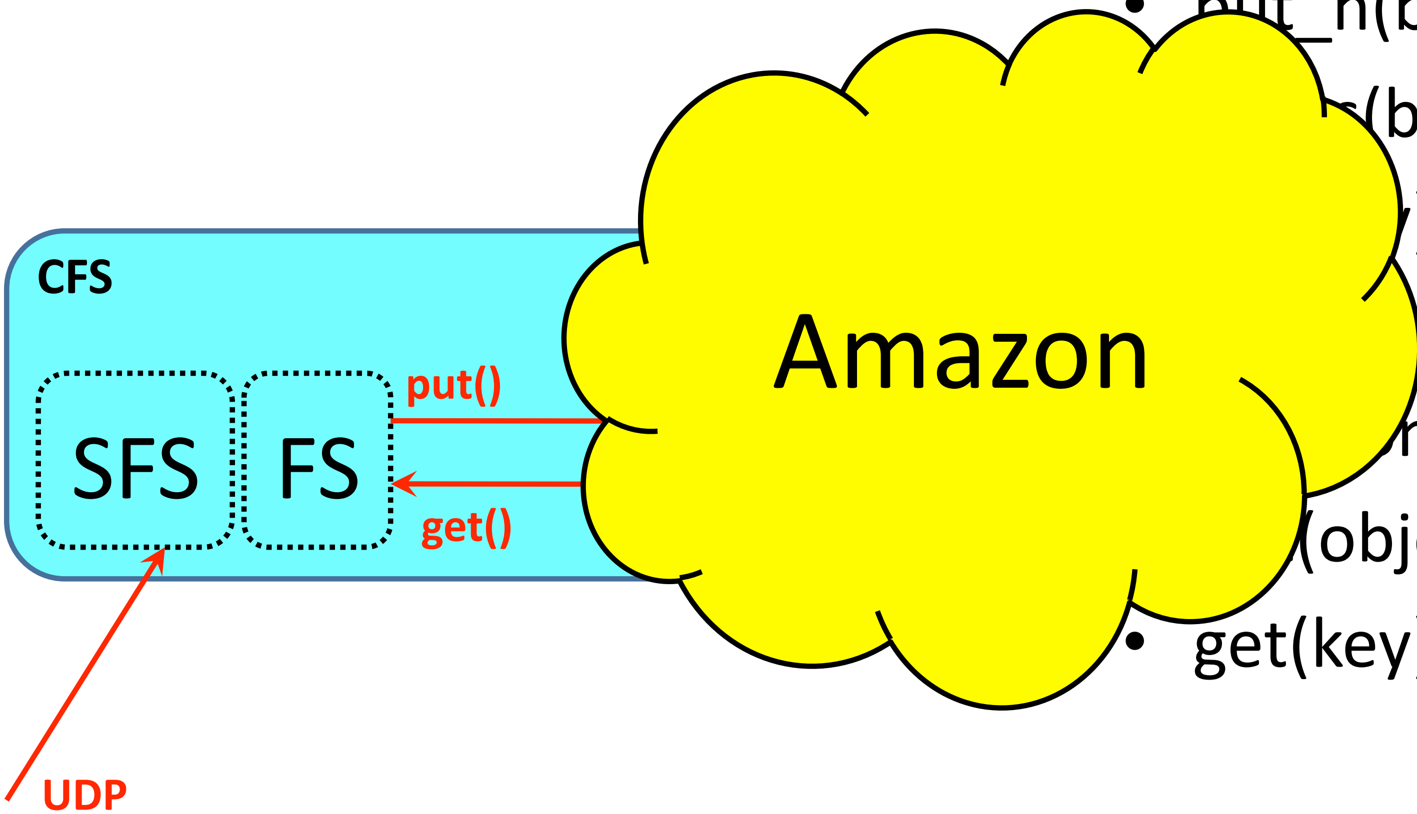
- Remember the files’ keys
- Use a directory block
- => Direcotry Key = H(D)

Root Block & Consistency

- Where to store the root key?
- Hash => update external references
- Append a signature and store at H(pubkey)
- Consistency?
- Only if everyone see the same version of the block



put/get API

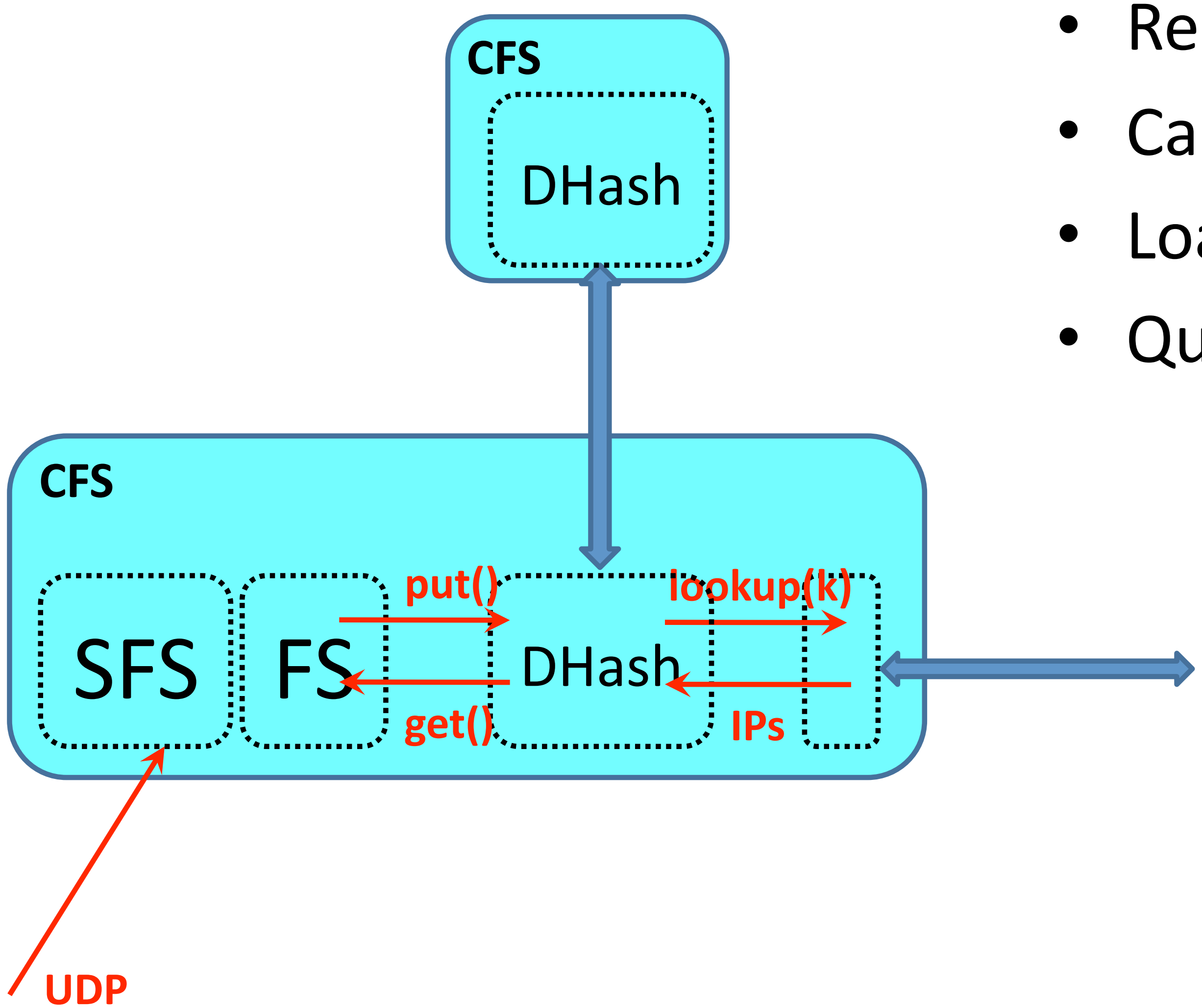


- put_h(block)
- s(block, pubkey)
- y)

n API:

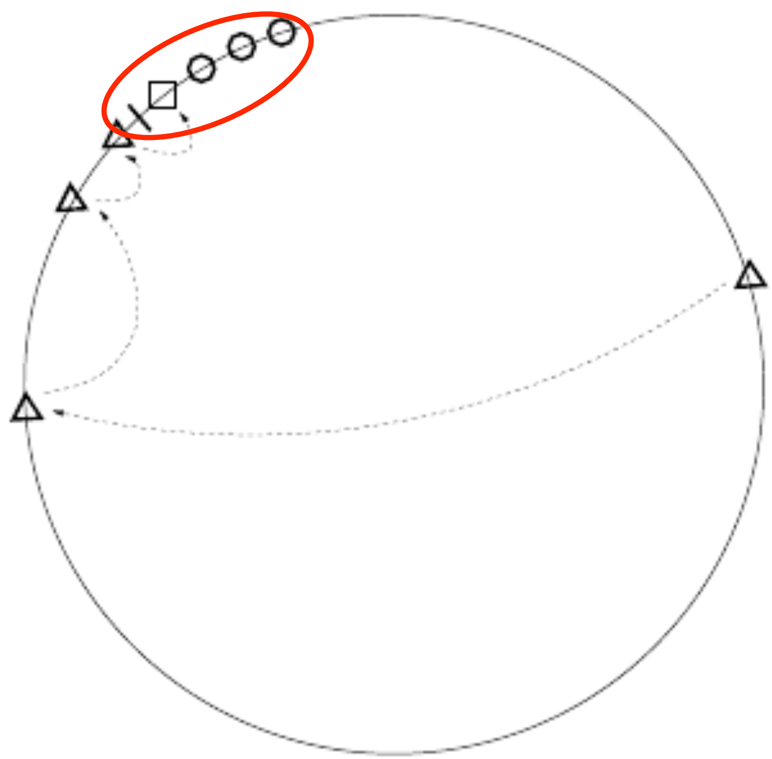
- (object,key)
- get(key)

DHash



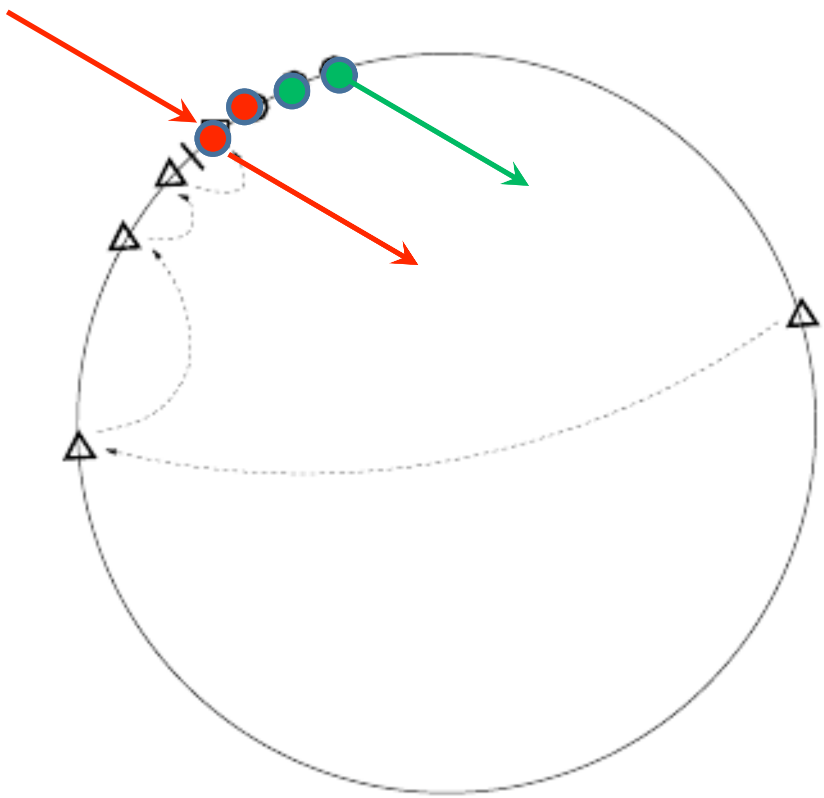
- Replication
- Caching
- Load balancing
- Quotas

Replication



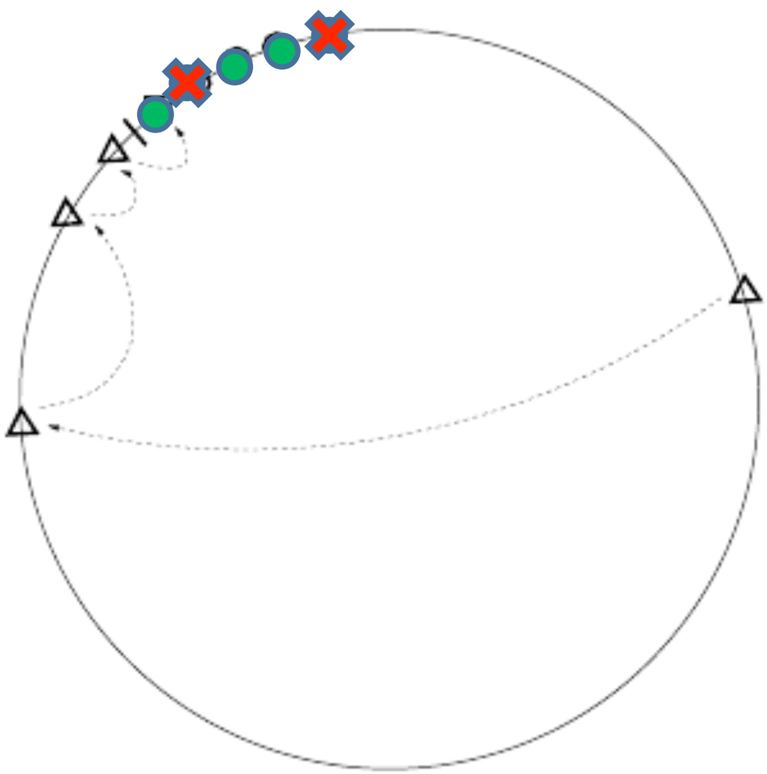
- Block (square) stored at the first successor of ID (thick mark)
- DHash maintains replicas on r successors given by Chord (circles)
- Each replicas are independent due to consistent hashing
- DHash do get() on one of the replicas depending on latency => load balancing

Replication



- Consistency ?
- put_h?
- put_s?

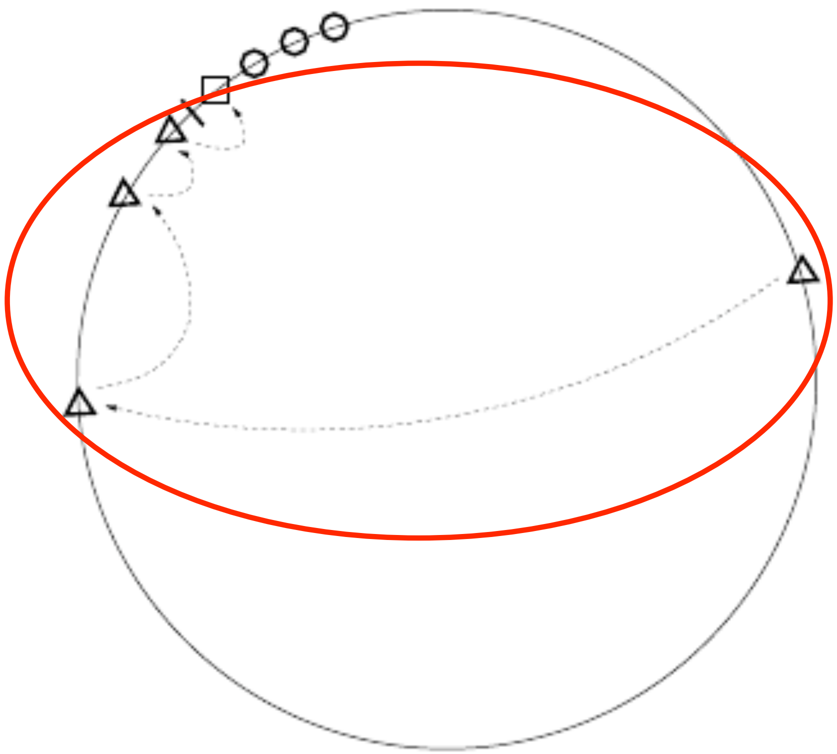
Transient & Permanent Failures



- Transient failures =>unnecessary uploads and deletes of replicas
- Lost of bandwidth

- Must remember the work done
- => have scope > r
- Decreases replication delay

Caching



- Caches blocks along the lookup path
- Least-recently-used replacement policy
- Keeps replicas close to the block ID

- Consistency?

Load balancing

- Consistent hashing => spread blocks across the ID space
- But variance would be log(N)
- And server capacities vary

- Introduce virtual servers: IDKey = Hash(IP||id)
- Security?
- Gives the freedom to chose id => dictionary attack => limit range of id => limit load-balancing

Quotas

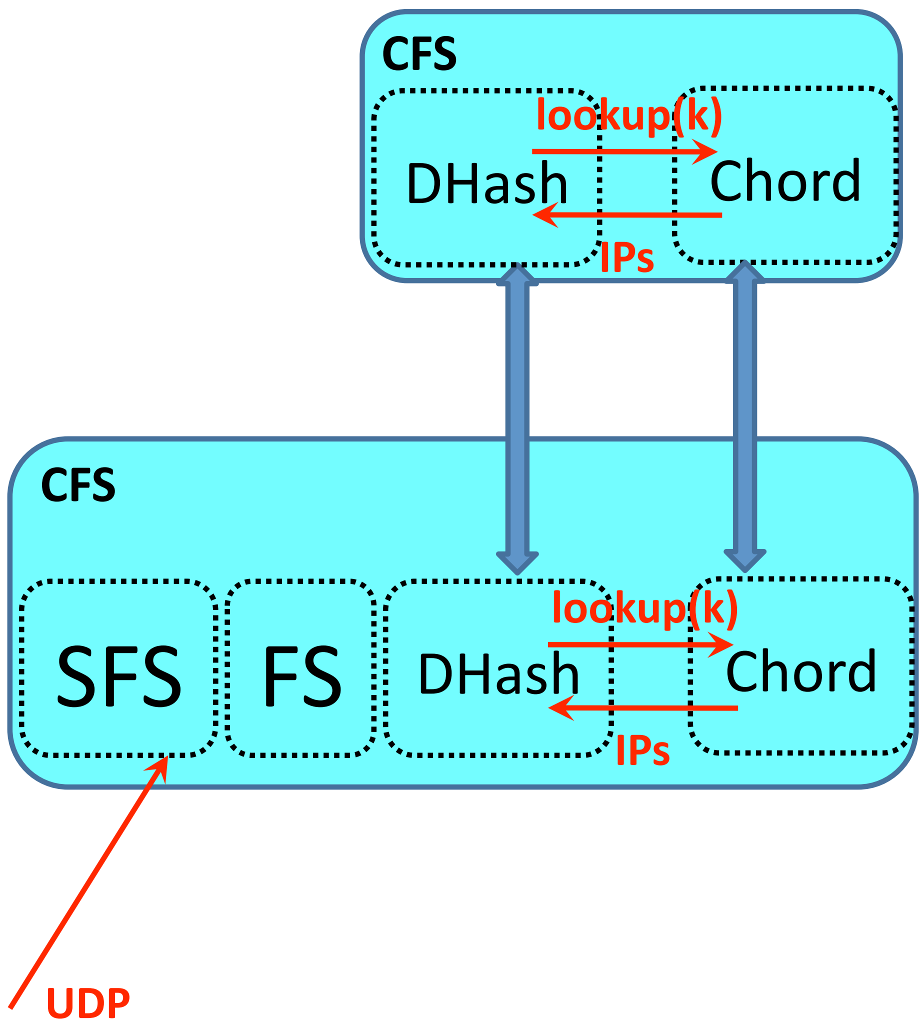
- Per-IP Quotas
- Does it work?
- Storage available is O(N) but storage provided is O(1) => Naturally overloaded
- Let's use O(1/N) quota
- But we need N*block size >> minimum storage space at a server

Updates and Deletions

- Only signed blocks can be updated
- Difficult to find a collision => prevents attack

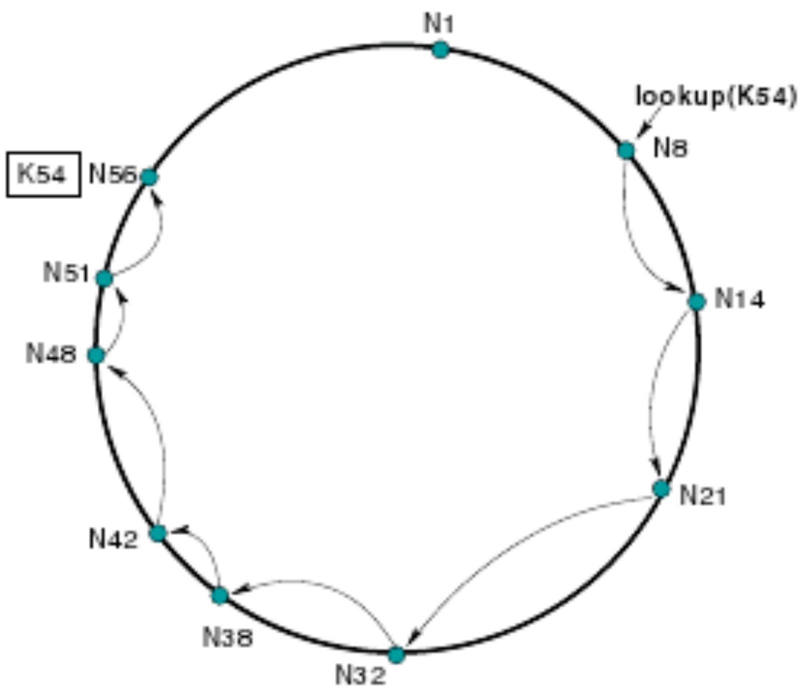
- No delete but refresh
- Recover from large amount of data inserted
- But consume bandwidth and loss-prone

Chord



- Key space is a ring, ie m = 0
- Associate each key to a list of r IPs
- Fault-tolerant
- Efficient: O(log(N))

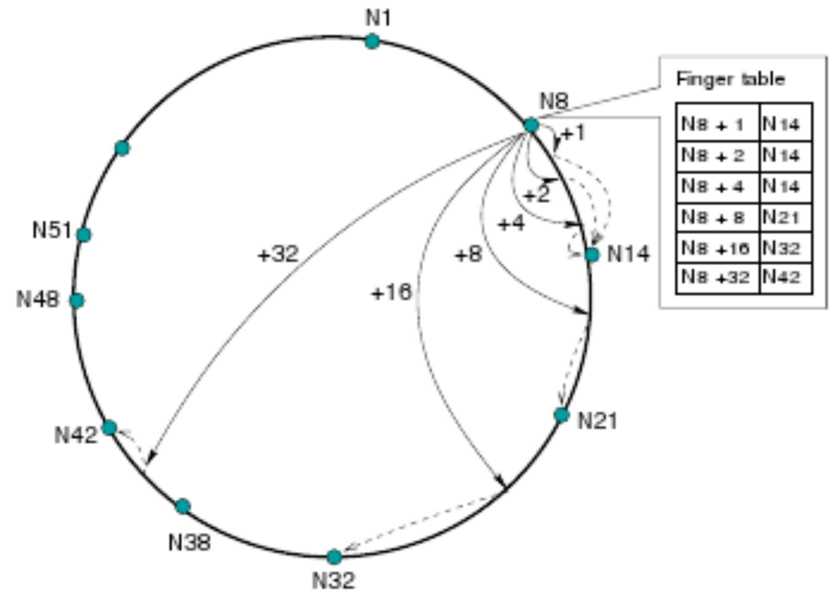
Linear Routing

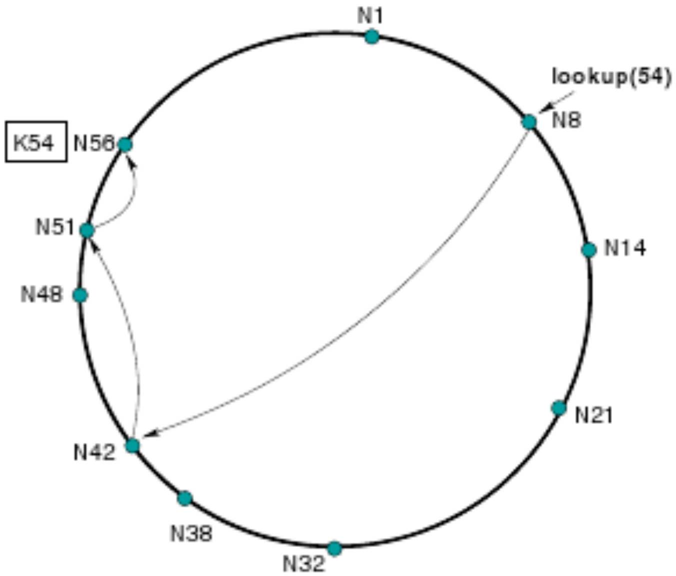



```
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n, n.successor])
        return n.successor;
    else
        // forward the query around the circle
        return successor.find_successor(id);
```

- Key and nodeID belongs to the same space
- Each key belongs to its successor node
- Always able to lookup a key if the list of successors is large enough
- Inefficient: O(N)

Recursive Routing

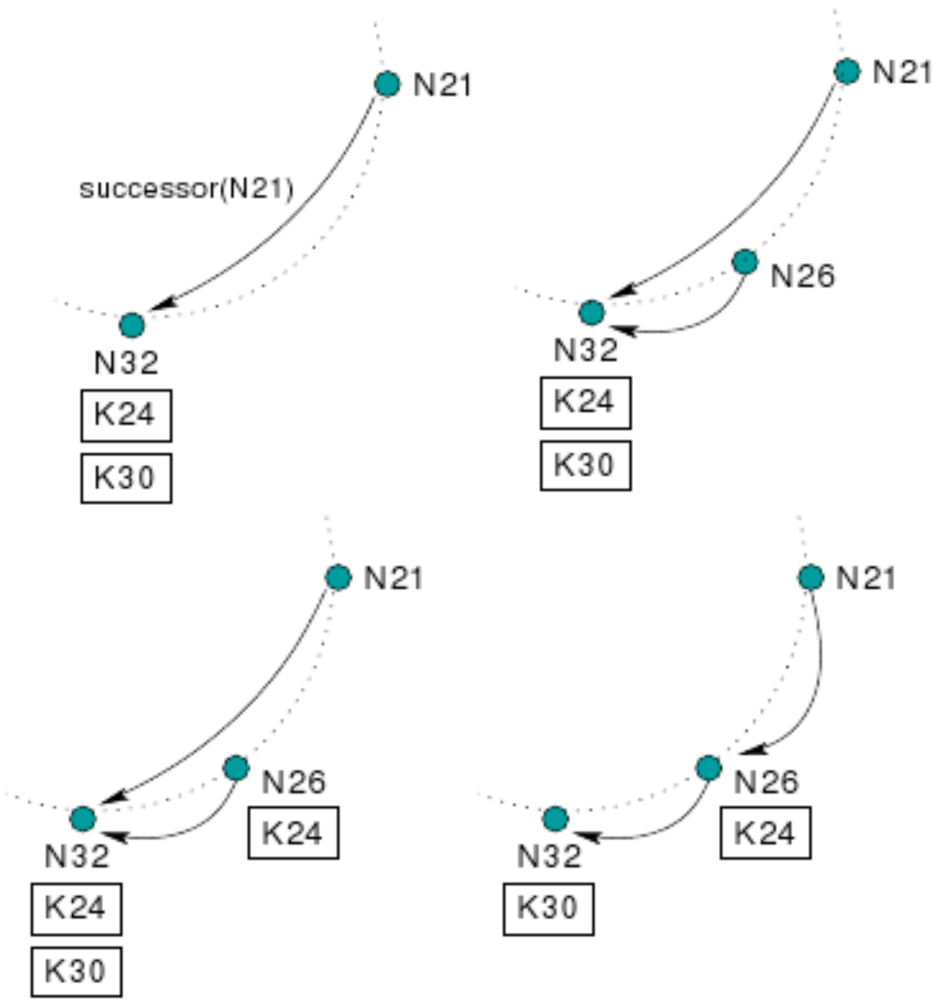




- Finger[k] = first node on circle that succeeds (n $+2^{k-1}$) mod 2m , 1 ≤ k ≤ m
- O(log(N)) in space

- O(log(N)) in hops
- Fingers tables aren't vital, just an accelerator

Joins



// periodically verify n's immediate successor,
// and tell the successor about n.
n.**stabilize**()
  $x = successor.predecessor;$
  if $(x \in (n, successor))$
    $successor = x;$
  $successor.notify(n);$

// n' thinks it might be our predecessor.
n.**notify**(n')
  if ($predecessor$ **is nil or** $n' \in (predecessor, n)$)
    $predecessor = n';$

Server Selection

- log(N): number of significant bits
- ones(): number of bit set to 1
- $H(n_i)$: estimation of the number of hops
- D: latencies

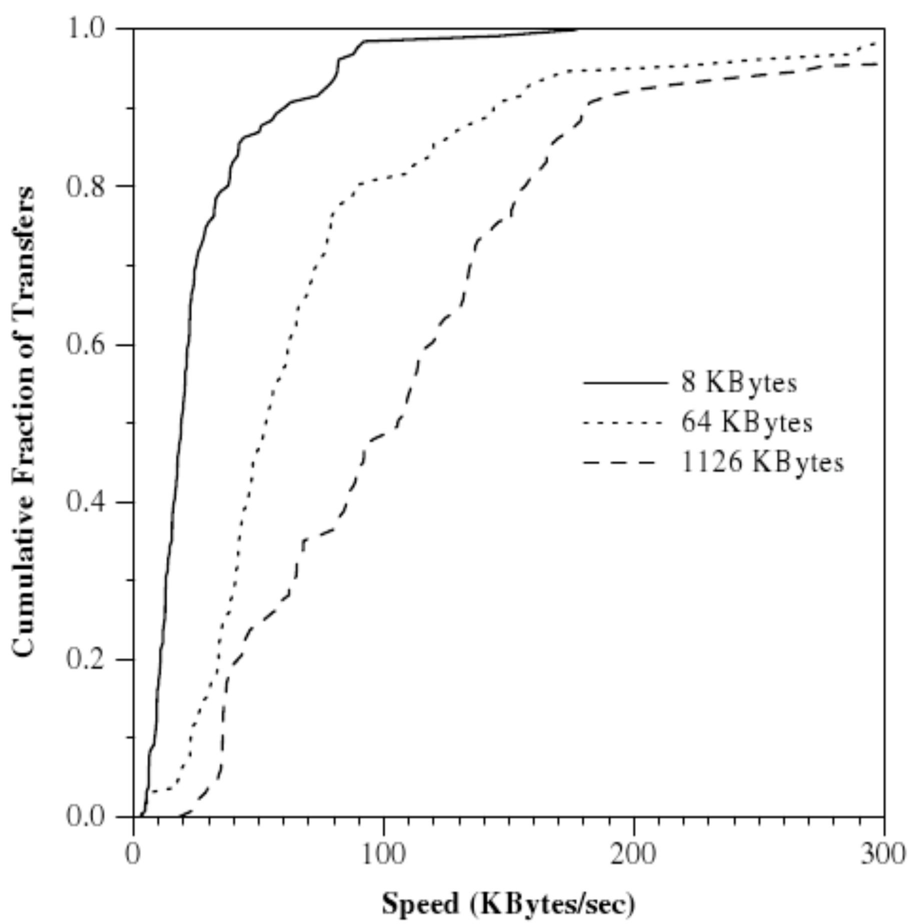
$$C(n_i) = d_i + \overline{d} \times H(n_i)$$
$$H(n_i) = \text{ones}((n_i - id) >> (160 - \log N))$$

Discussion

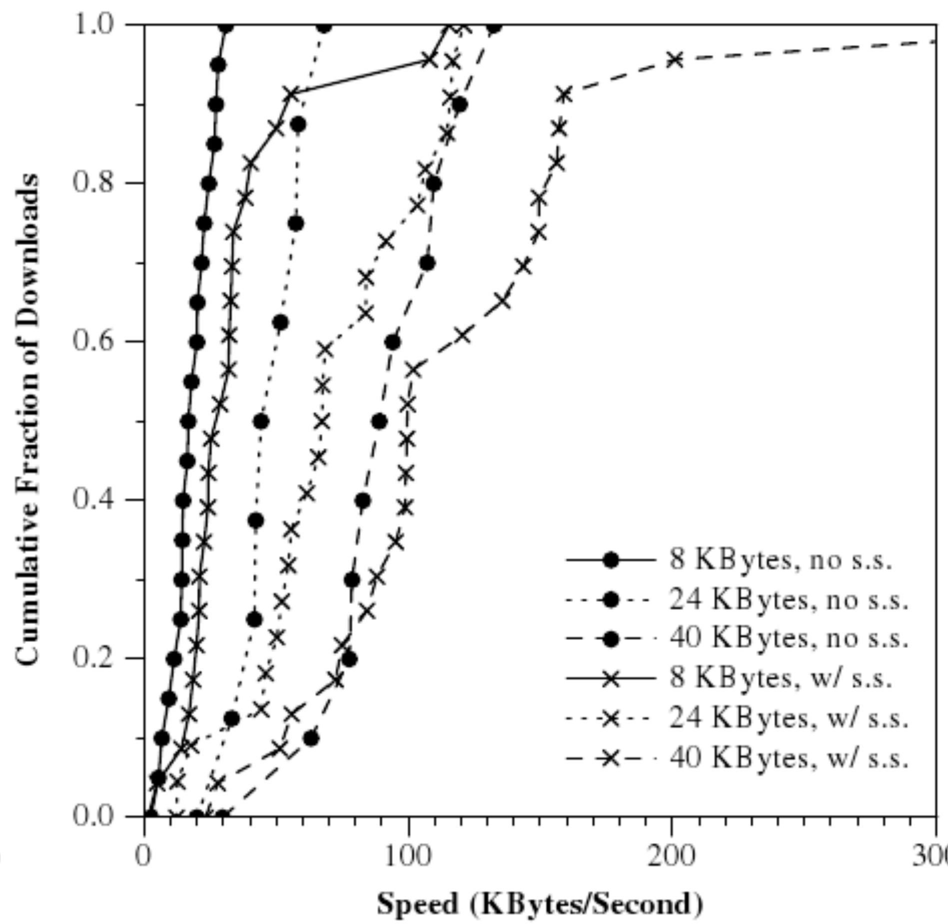
- Efficiency
- Load Balance
- Persistence
- Decentralized control
- Scalability
- Availability
- Quotas
- Consistency ?

Efficiency

FTP – different file sizes

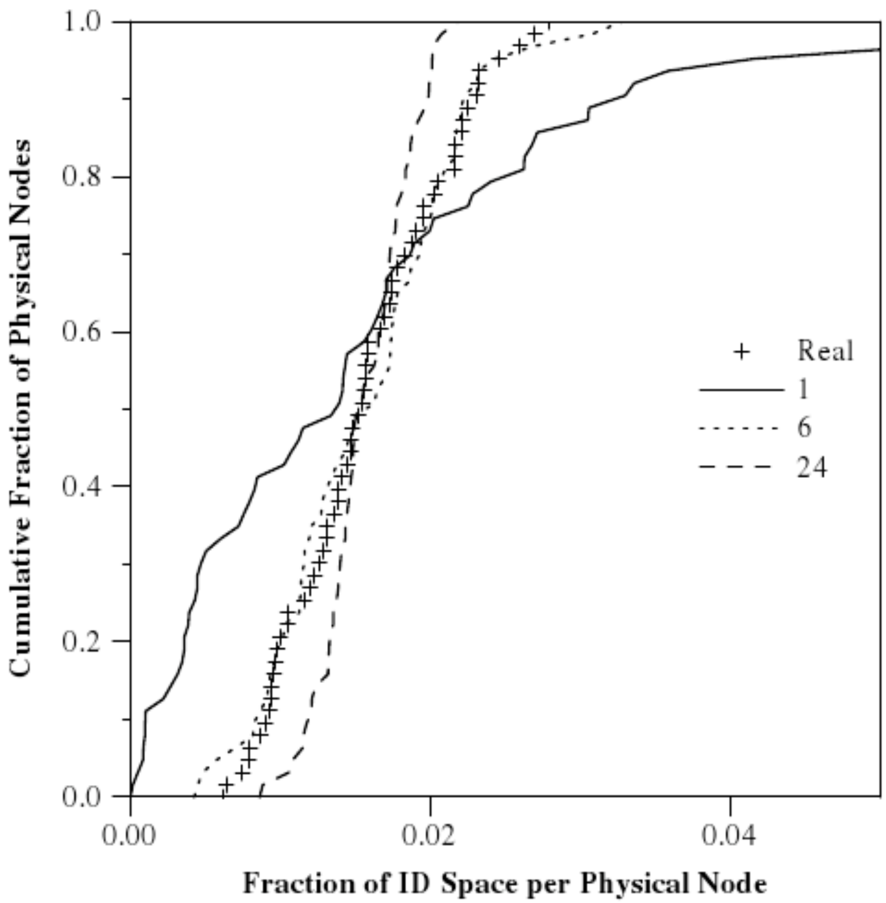


CFS – w/ and w/o server selection

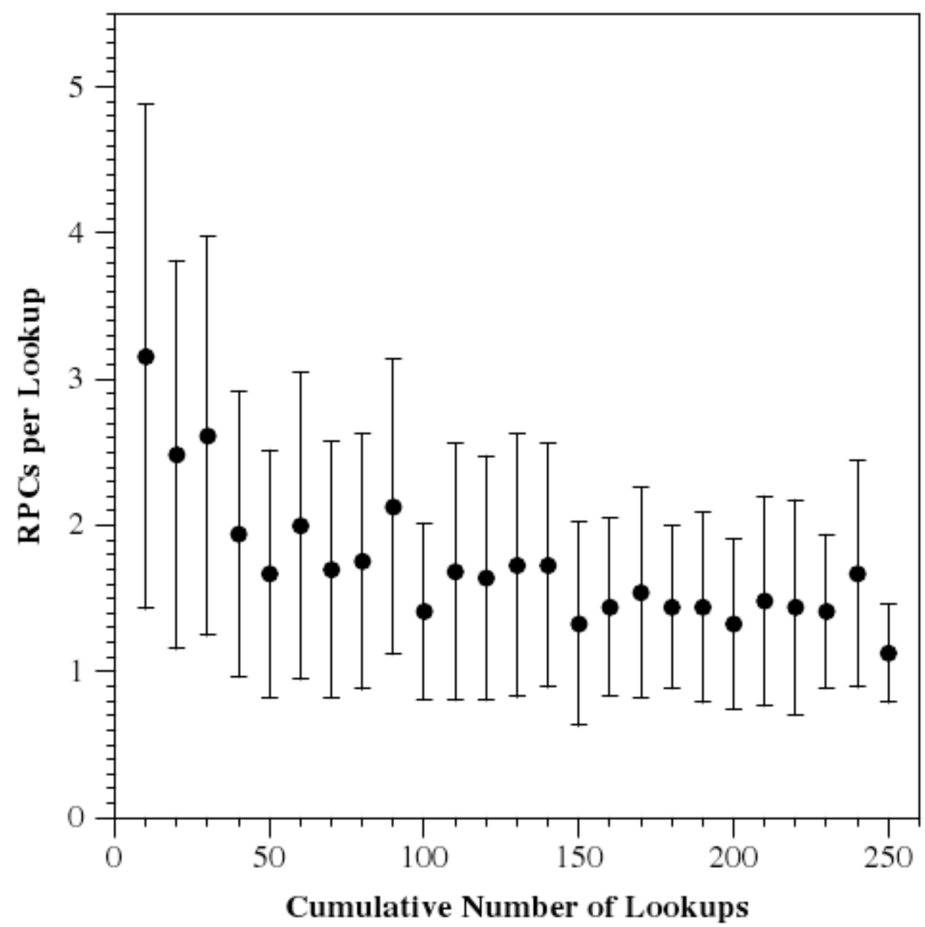


Load Balancing

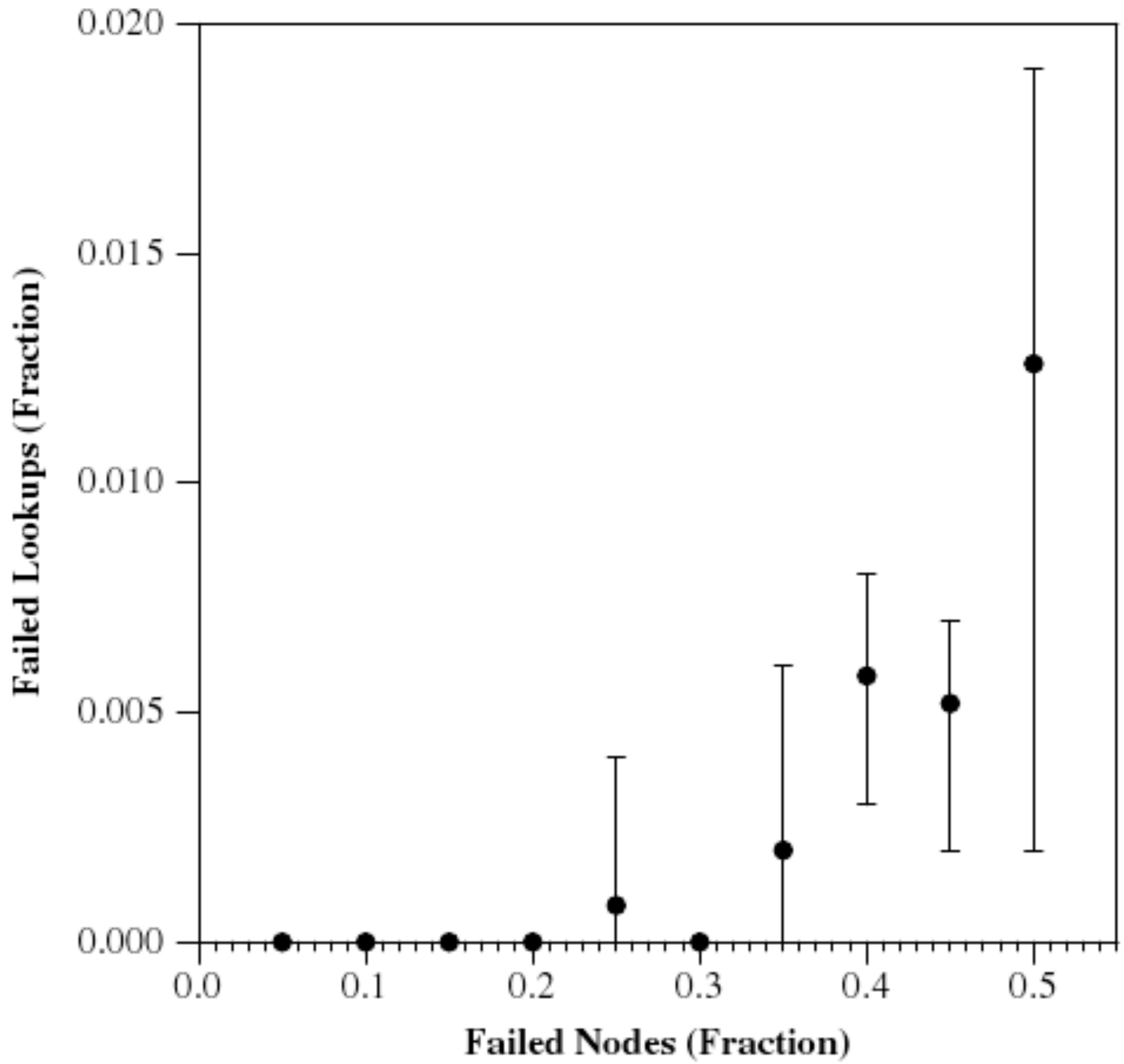
Storage – 64 servers



Banwidth – 1000 servers



Persistence



Discussion

- Decentralized control ?
- Scalability ?
- Availability ?
- Quotas ?
- Consistency ?