CS 268: Lecture 22
DHT Applications

Ion Stoica
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1776

(Presentation based on slides from Robert Morris and Sean Rhea)

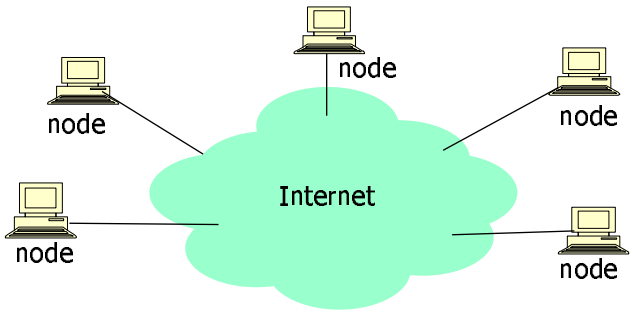
1

Outline

- Cooperative File System (CFS)
- Open DHT

2

Target CFS Uses



- Serving data with inexpensive hosts:
  - open-source distributions
  - off-site backups
  - tech report archive
  - efficient sharing of music

3

How to mirror open-source distributions?

- Multiple independent distributions
  - Each has high peak load, low average
- Individual servers are wasteful
- Solution: aggregate
  - Option 1: single powerful server
  - Option 2: distributed service
    - But how do you find the data?

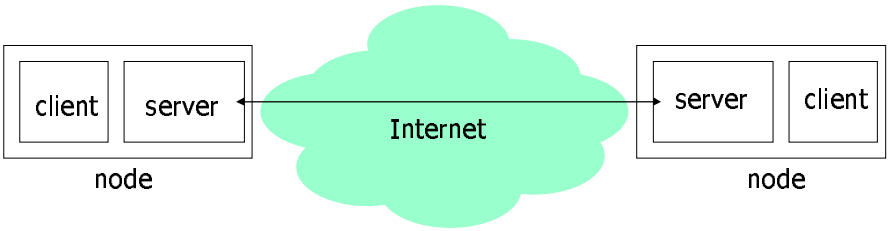
4

Design Challenges

- Avoid hot spots
- Spread storage burden evenly
- Tolerate unreliable participants
- Fetch speed comparable to whole-file TCP
- Avoid O(#participants) algorithms
  - Centralized mechanisms [Napster], broadcasts [Gnutella]
- CFS solves these challenges

5

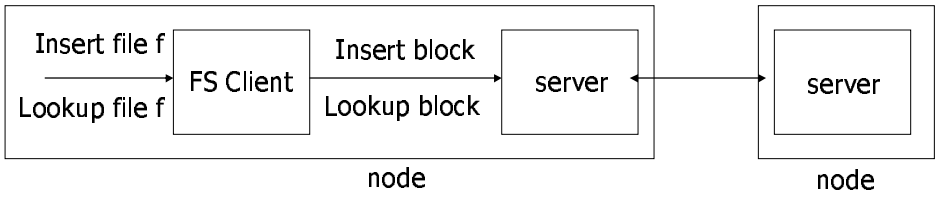
CFS Architecture



- Each node is a client and a server
- Clients can support different interfaces
  - File system interface
  - Music key-word search

6

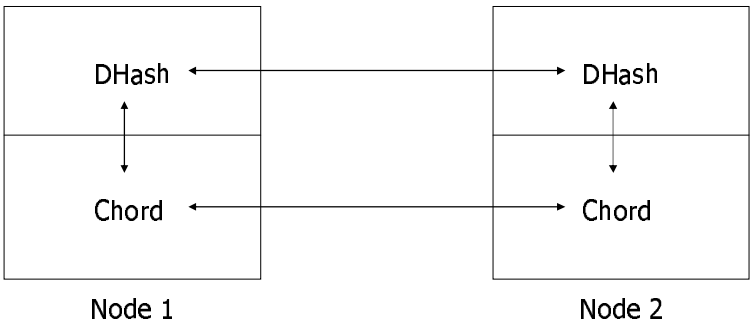
Client-server interface



- Files have unique names
- Files are read-only (single writer, many readers)
- Publishers split files into blocks
- Clients check files for authenticity

7

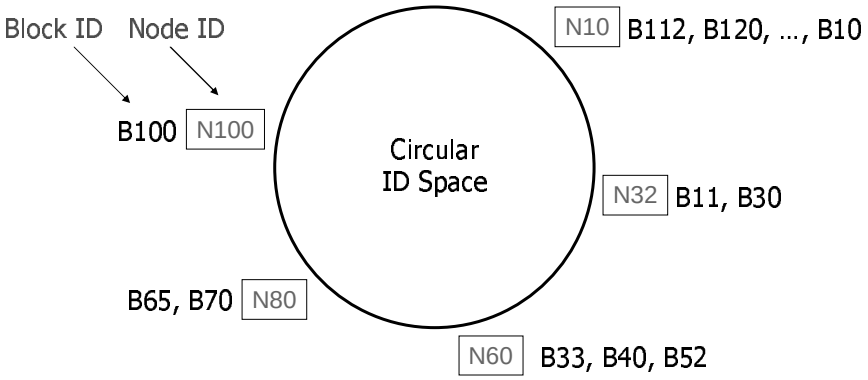
Server Structure



- DHash stores, balances, replicates, caches blocks
- DHash uses Chord [SIGCOMM 2001] to locate blocks

8

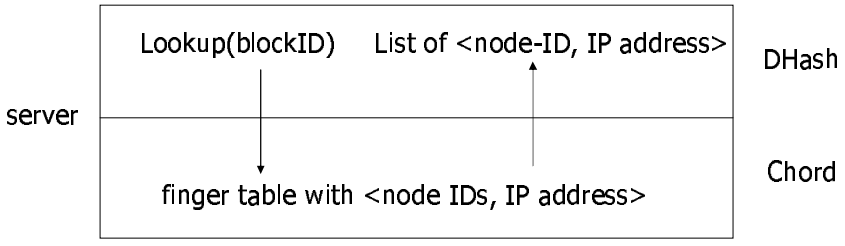
Chord Hashes a Block ID to its *Successor*



- Nodes and blocks have randomly distributed IDs
- Successor: node with next highest ID

9

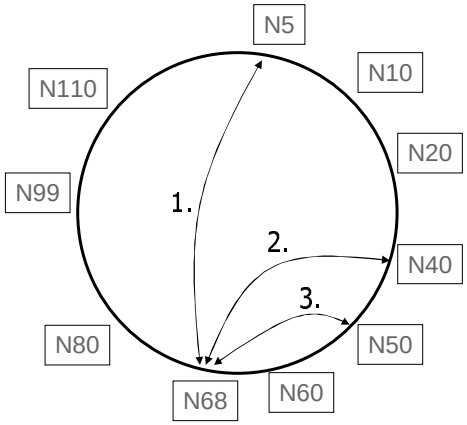
DHash/Chord Interface



- *lookup()* returns list with node IDs closer in ID space to block ID
  - Sorted, closest first

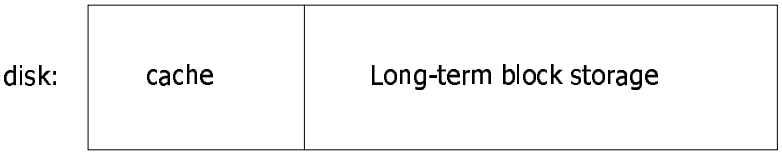
10

DHash Uses Other Nodes to Locate Blocks



11

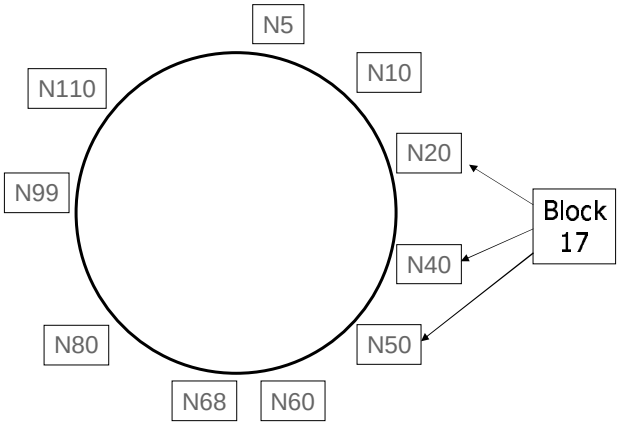
Storing Blocks



- Long-term blocks are stored for a fixed time
  - Publishers need to refresh periodically
- Cache uses LRU

12

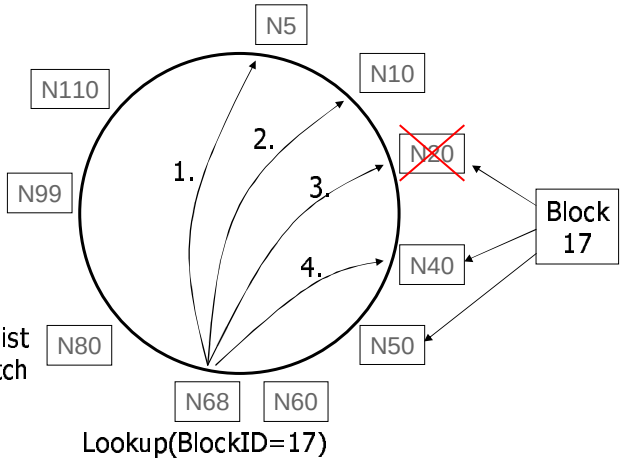
Replicate blocks at *r* successors



- Node IDs are SHA-1 of IP Address
- Ensures independent replica failure

13

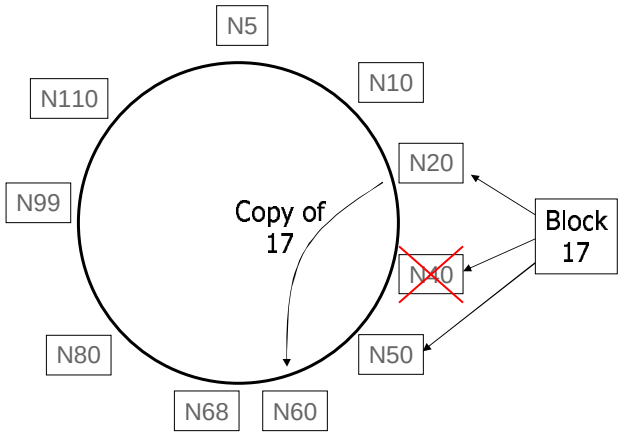
Lookups find replicas



RPCs:
1. Lookup step
2. Get successor list
3. Failed block fetch
4. Block fetch

14

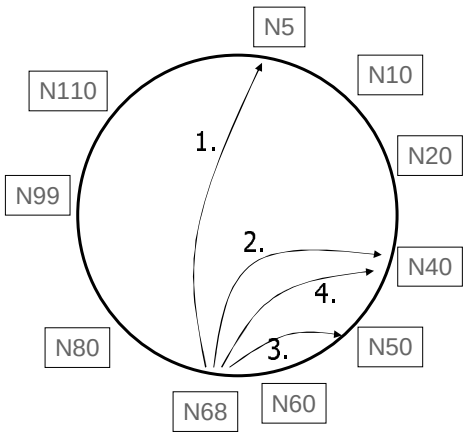
First Live Successor Manages Replicas



- Node can locally determine that it is the first live successor

15

DHash Copies to Caches Along Lookup Path

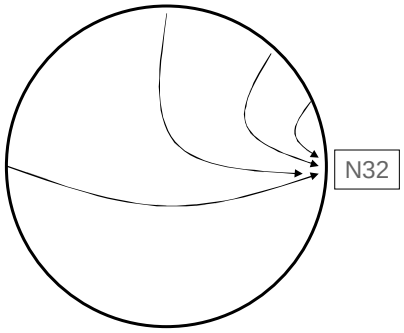


RPCs:
1. Chord lookup
2. Chord lookup
3. Block fetch
4. Send to cache

Lookup(BlockID=45)

16

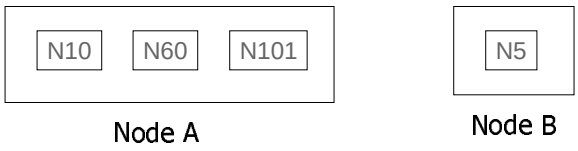
Caching at Fingers Limits Load



- Only O(log N) nodes have fingers pointing to N32
- This limits the single-block load on N32

17

Virtual Nodes Allow Heterogeneity



Node A Node B

- Hosts may differ in disk/net capacity
- Hosts may advertise multiple IDs
  - Chosen as SHA-1(IP Address, index)
  - Each ID represents a "virtual node"
- Host load proportional to # v.n.'s
- Manually controlled

18

Why Blocks Instead of Files?

- Cost: one lookup per block
  - Can tailor cost by choosing good block size
- Benefit: load balance is simple
  - For large files
  - Storage cost of large files is spread out
  - Popular files are served in parallel

19

Outline

- Cooperative File System (CFS)
- Open DHT

20

Questions:

- How many DHTs will there be?
- Can all applications share one DHT?

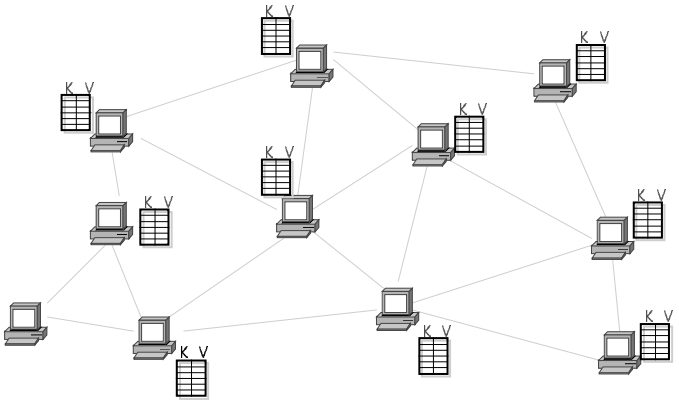
21

Benefits of Sharing a DHT

- Amortizes costs across applications
  - Maintenance bandwidth, connection state, etc.
- Facilitates “bootstrapping” of new applications
  - Working infrastructure already in place
- Allows for statistical multiplexing of resources
  - Takes advantage of spare storage and bandwidth
- Facilitates upgrading existing applications
  - “Share” DHT between application versions

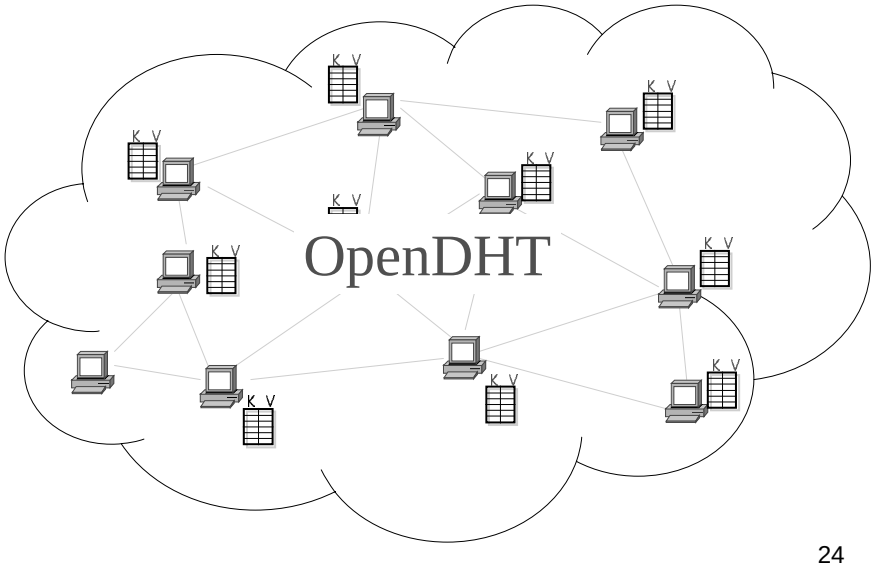
22

The DHT as a Service



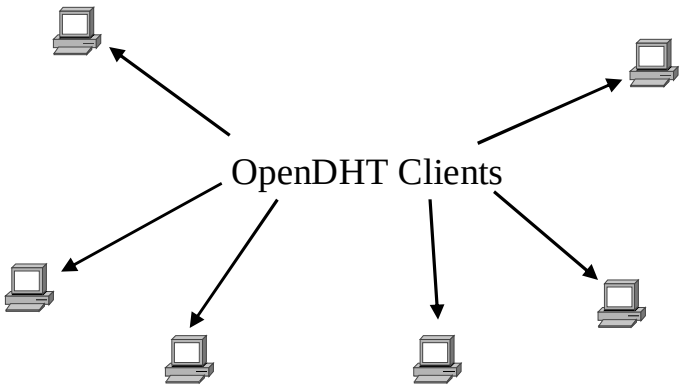
23

The DHT as a Service



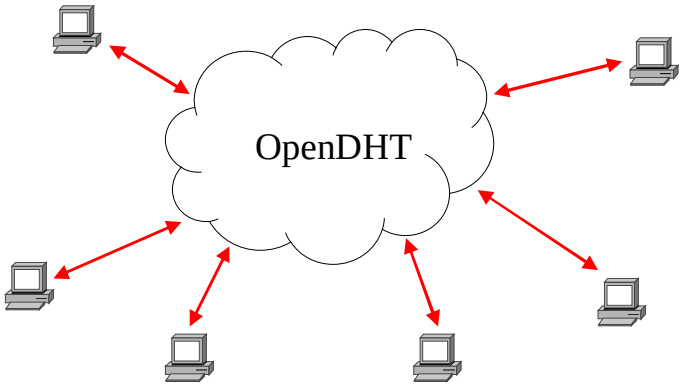
24

The DHT as a Service



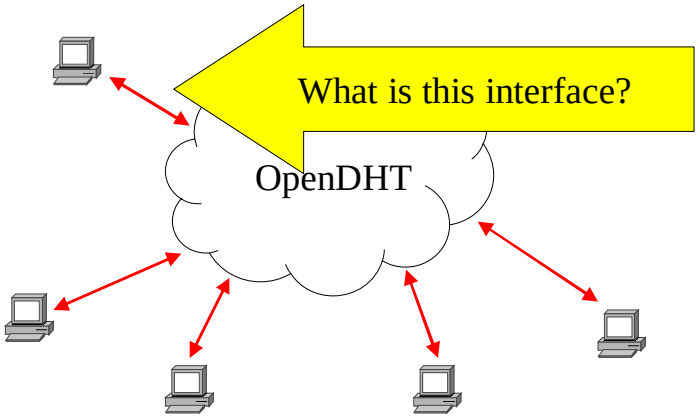
25

The DHT as a Service



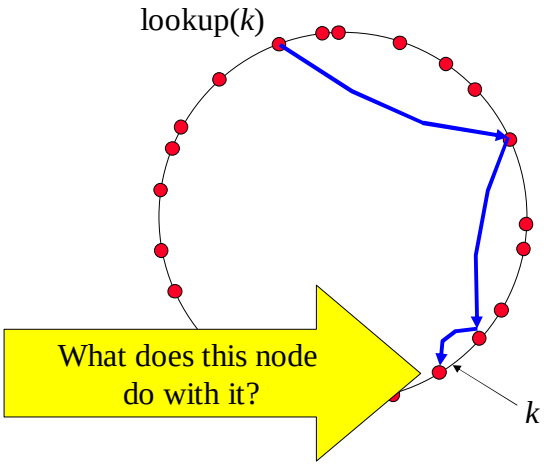
26

The DHT as a Service



27

It's not lookup()



Challenges:
1. Distribution
2. Security

28

How are DHTs Used?

1. Storage
   - CFS, UsenetDHT, PKI, etc.

2. Rendezvous
   - Simple: Chat, Instant Messenger
   - Load balanced: *i*3
   - Multicast: RSS Aggregation, White Board
   - Anycast: Tapestry, Coral

29

What about put/get?

- Works easily for storage applications
- Easy to share
  - No upcalls, so no code distribution or security complications
- But does it work for rendezvous?
  - Chat? Sure: put(my-name, my-IP)
  - What about the others?

30

Protecting Against Overuse

- Must protect system resources against overuse
  - Resources include network, CPU, and disk
  - Network and CPU straightforward
  - Disk harder: usage persists long after requests
- Hard to distinguish malice from eager usage
  - Don't want to hurt eager users if utilization low
- Number of active users changes over time
  - Quotas are inappropriate

31

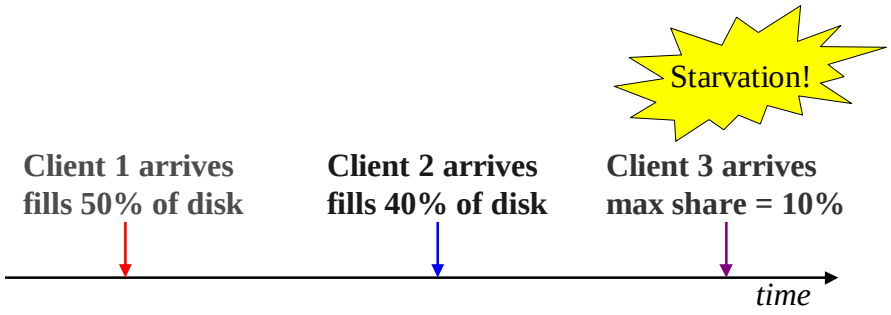
Fair Storage Allocation

- Our solution: give each client a fair share
  - Will define "fairness" in a few slides
- Limits strength of malicious clients
  - Only as powerful as they are numerous
- Protect storage on each DHT node separately
  - Must protect each subrange of the key space
  - Rewards clients that balance their key choices

32

The Problem of Starvation

- Fair shares change over time
  - Decrease as system load increases



33

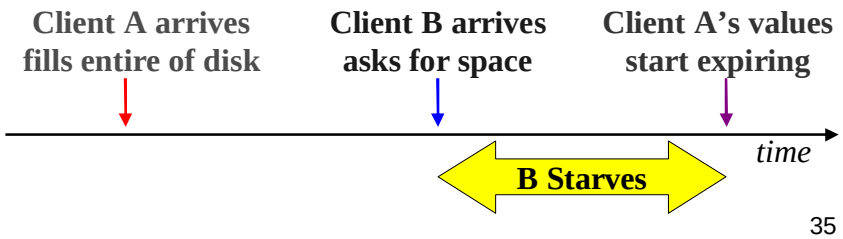
Preventing Starvation

- Simple fix: add time-to-live (TTL) to puts
  - put (key, value) → put (key, value, ttl)
- Prevents long-term starvation
  - Eventually all puts will expire

34

Preventing Starvation

- Simple fix: add time-to-live (TTL) to puts
  - put (key, value) → put (key, value, ttl)
- Prevents long-term starvation
  - Eventually all puts will expire
- Can still get short term starvation

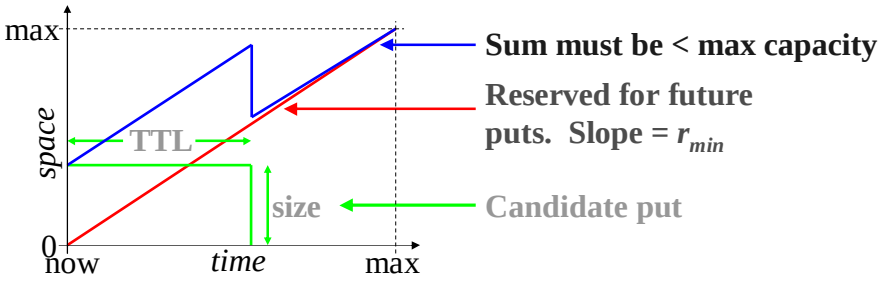


35

Preventing Starvation

- Stronger condition:

  Be able to accept r_{min} bytes/sec new data at all times
- This is non-trivial to arrange!



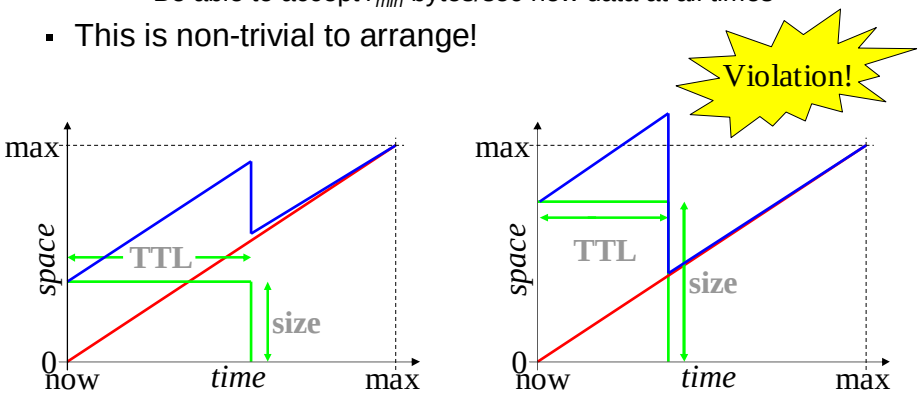
36

Preventing Starvation

- Stronger condition:

  Be able to accept r_{min} bytes/sec new data at all times

- This is non-trivial to arrange!



37

Preventing Starvation

- Formalize graphical intuition:

  $f(\tau) = B(t_{now}) - D(t_{now}, t_{now}+\tau) + r_{min} \times \tau$

  - $D(t_{now}, t_{now}+\tau)$: aggregate size of puts expiring in the interval $(t_{now}, t_{now}+\tau)$

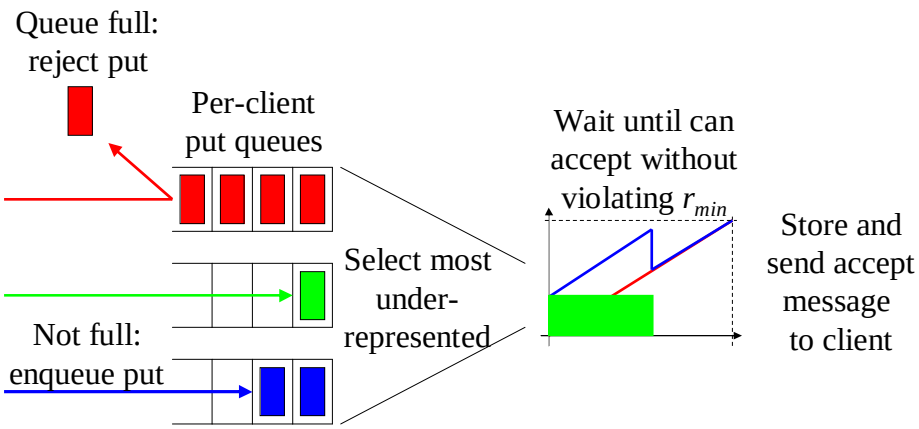
- To accept put of size x and TTL l:

  $f(\tau) + x < C \quad$ for all $0 \le \tau < l$

- Can track the value of f efficiently with a tree
  - Leaves represent inflection points of f
  - Add put, shift time are O(log n), n = # of puts

38

Fair Storage Allocation

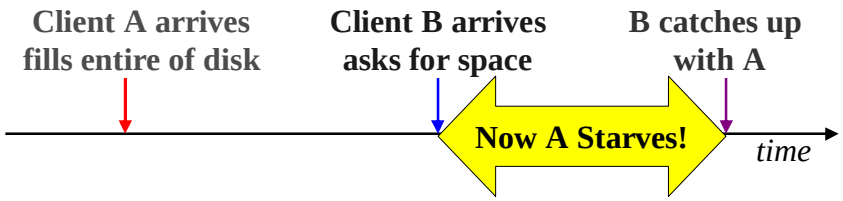


The Big Decision: Definition of "most under-represented"

39

Defining "Most Under-Represented"

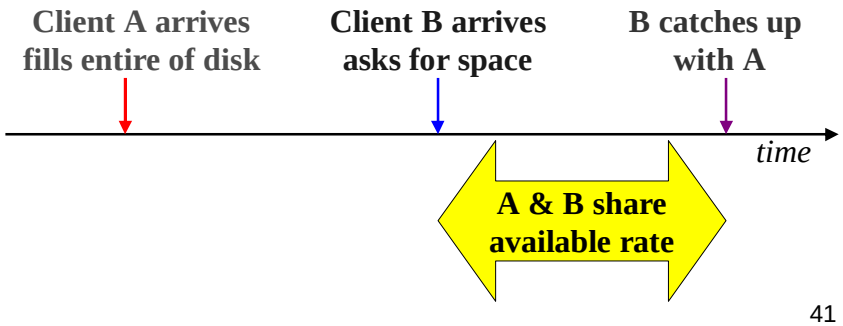
- Not just sharing disk, but disk over time
  - 1 byte put for 100s same as 100 byte put for 1s
  - So units are bytes × seconds, call them *commitments*
- Equalize total commitments granted?
  - No: leads to starvation
  - A fills disk, B starts putting, A starves up to max TTL



40

Defining "Most Under-Represented"

- Instead, equalize *rate* of commitments granted
  - Service granted to one client depends only on others putting "at same time"



41

Defining "Most Under-Represented"

- Instead, equalize *rate* of commitments granted
  - Service granted to one client depends only on others putting "at same time"
- Mechanism inspired by Start-time Fair Queuing
  - Have virtual time, v(t)
  - Each put gets a start time $S(p_c^i)$ and finish time $F(p_c^i)$

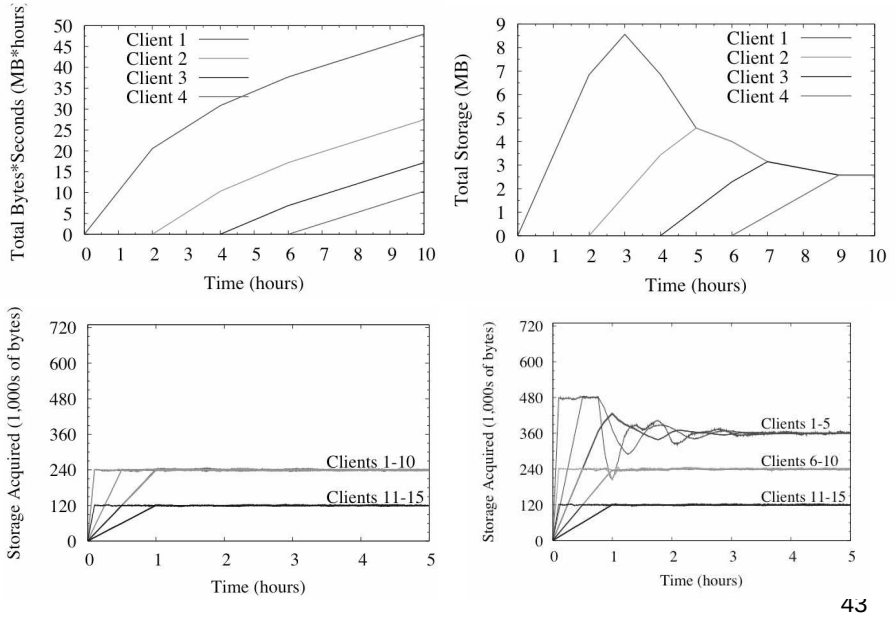
$$F(p_c^i) = S(p_c^i) + size(p_c^i) \times ttl(p_c^i)$$

$$S(p_c^i) = \max(v(A(p_c^i)) - \varepsilon, F(p_c^{i-1}))$$

$$v(t) = \text{maximum start time of all accepted puts}$$

42

FST Performance



43