# Shark: Scaling File Servers via Cooperative Caching

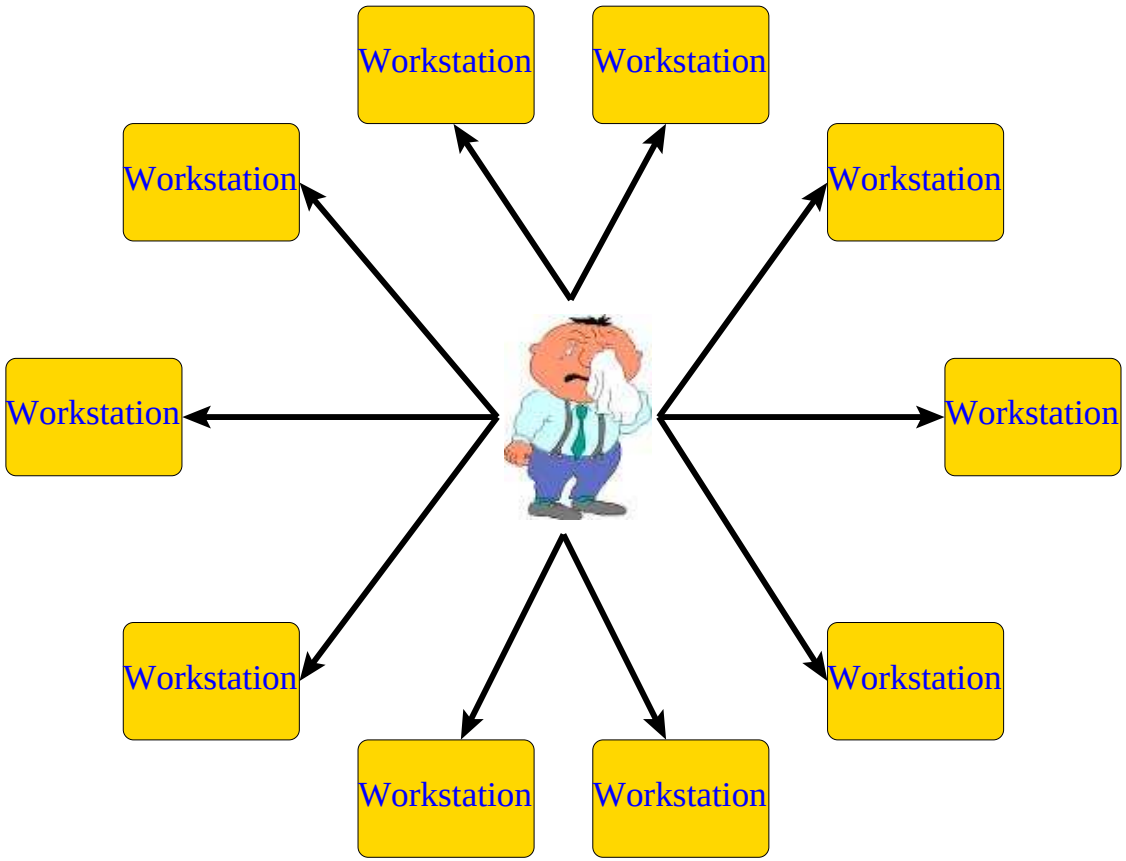
Siddhartha Annapureddy  Michael J. Freedman  David Mazières

New York University

Networked Systems Design and Implementation, 2005

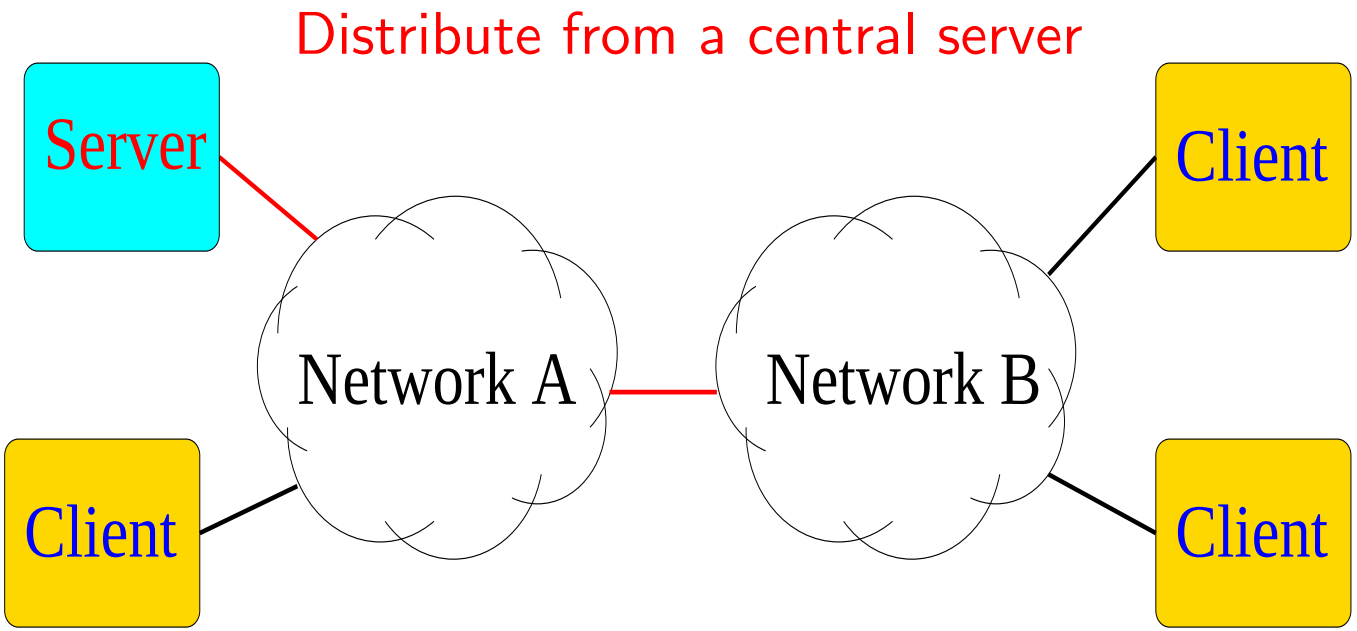
# Motivation

Scenario – Want to test a new application on a hundred nodes




Problem – Need to push software to all nodes and collect results

# Current Approaches

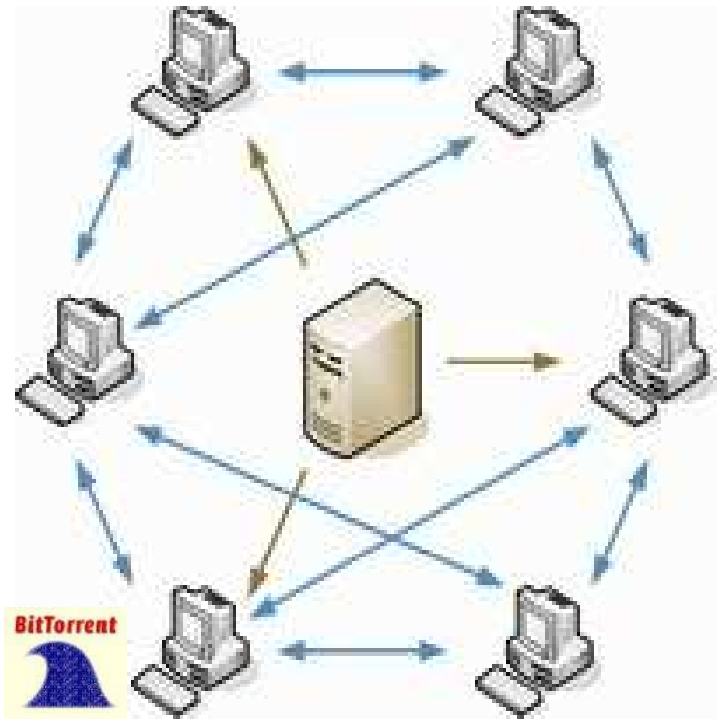



`rsync, unison, scp`

- Server's network uplink saturates
  - Operation takes a long time to finish
- Wastes bandwidth along bottleneck links

# Current Approaches

File distribution mechanisms




`BitTorrent, Bullet`

+ Scales by offloading burden on server
– Client downloads from half-way across the world

# Inherent Problems with Copying Files

**Users have to decide a priori what to ship**

- Ship too much – Waste bandwidth, takes long time
- Ship too little – Hassle to work in a poor environment

**Idle environments consume disk space**

- Users are loath to cleanup $\Rightarrow$ Redistribution
- Need low cost solution for refetching files

**Manual management of software versions**

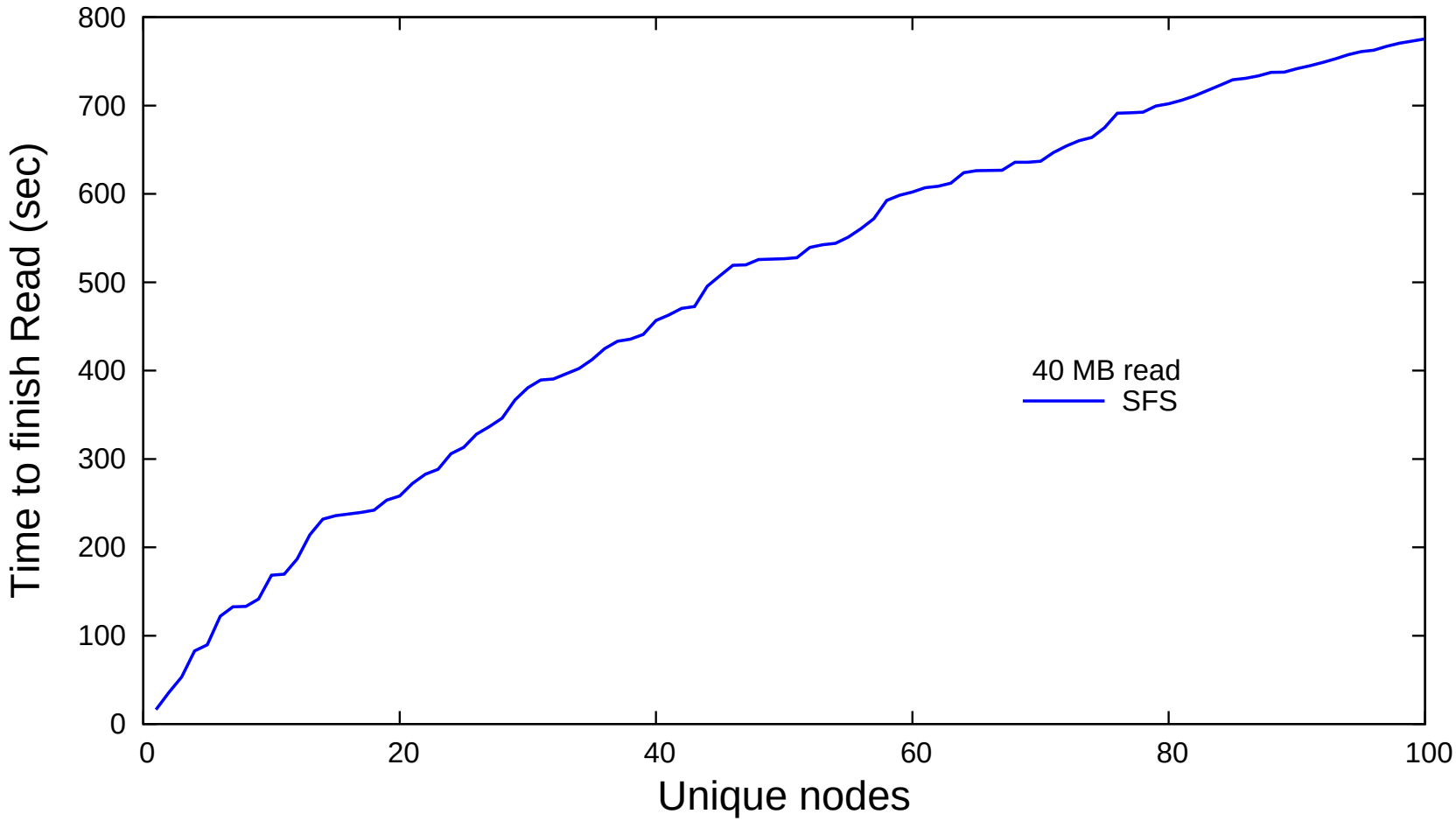
Illusion of having development environment
Programs fetched transparently on demand

# Networked File Systems

+ Know how to deploy these systems
+ Know how to administer such systems
+ Simple accountability mechanisms

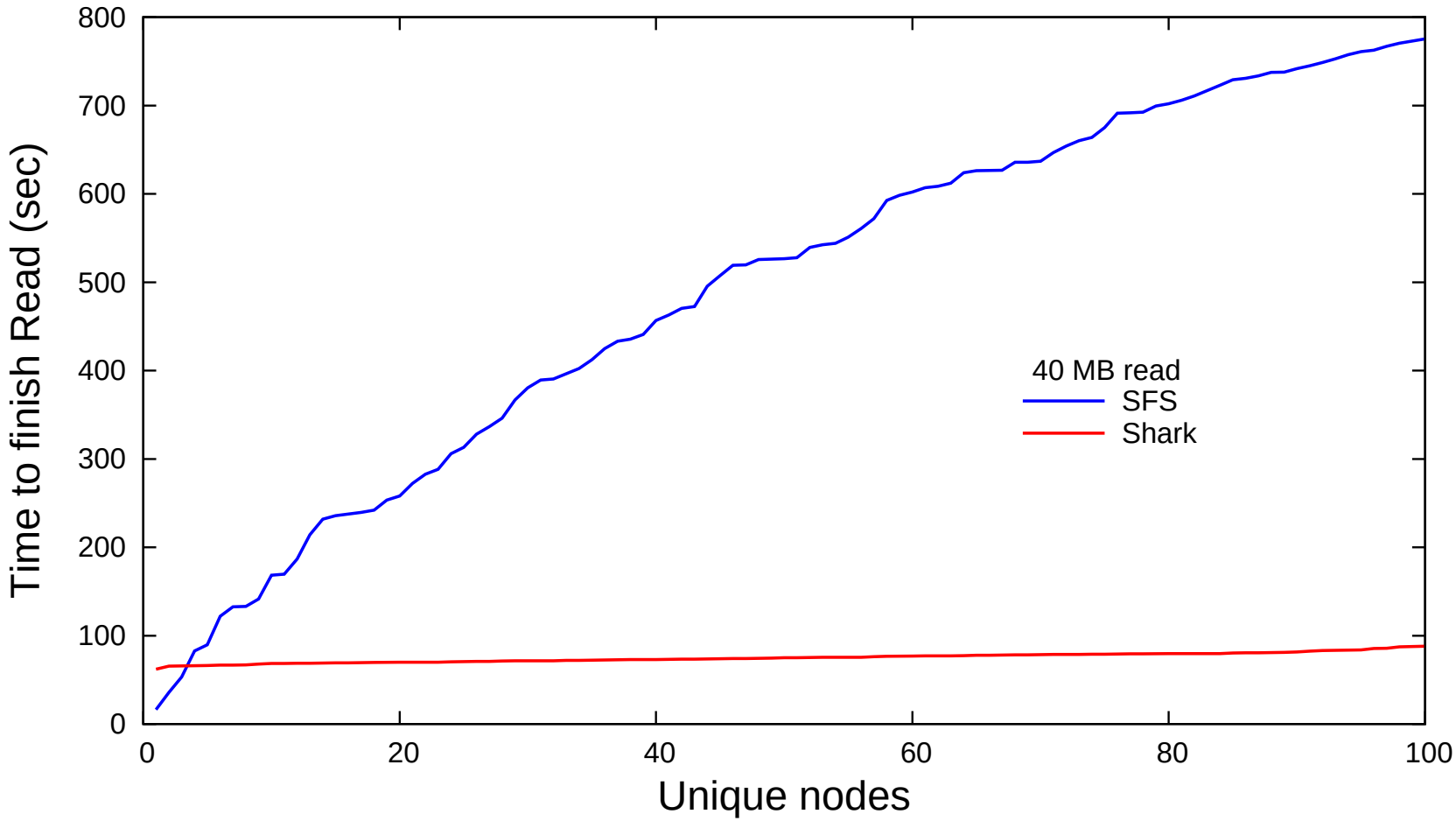
**Eg:** NFS, AFS, SFS etc.

# Problem: Scalability




Very slow $\approx$ 775s

# Problem: Scalability




Very slow $\approx$ 775s

Much better !!! $\approx$ 88s (9x better)

# Peer-to-Peer Filesystems

**P2P Filesystems (Pond, Ivy)**

+ Scalability
- Non-standard administrative models
- New filesystem semantics
- Haven't been widely deployed yet

Goal – Best of Both Worlds

- Convenience of central servers
- Scalability of peer-to-peer

# Let's Design this New Filesystem


**Vanilla SFS**

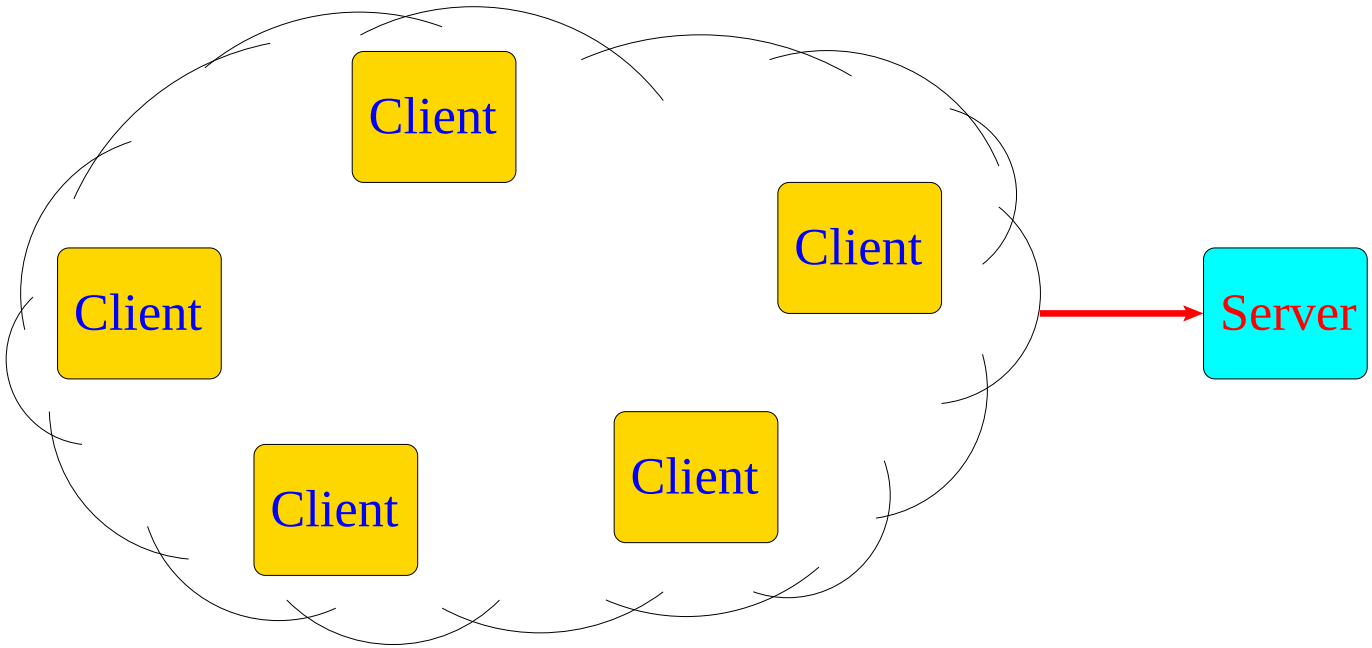
- Central server model

# Scalability – Client-side Caching


Large client cache à la AFS

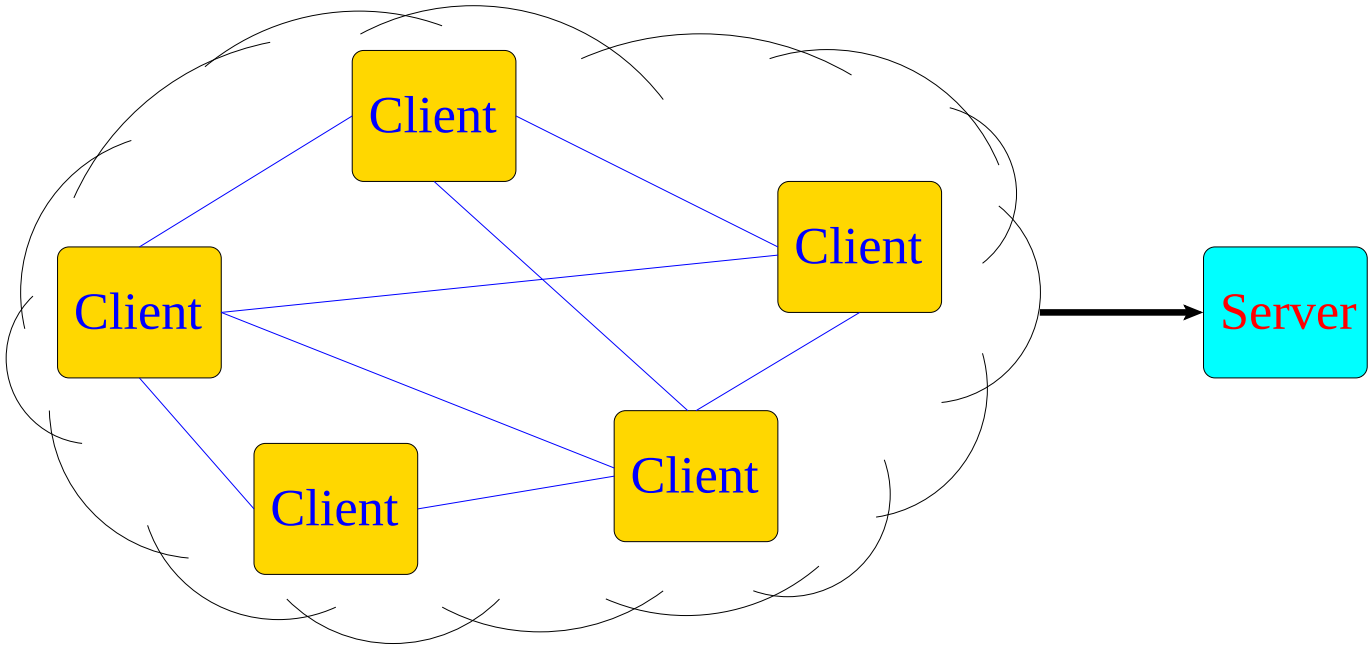
- Scales by avoiding redundant data transfers
- Leases to ensure NFS-style cache consistency
- With multiple clients, must address bandwidth concerns

# Scalability – Cooperative Caching



# Scalability – Cooperative Caching

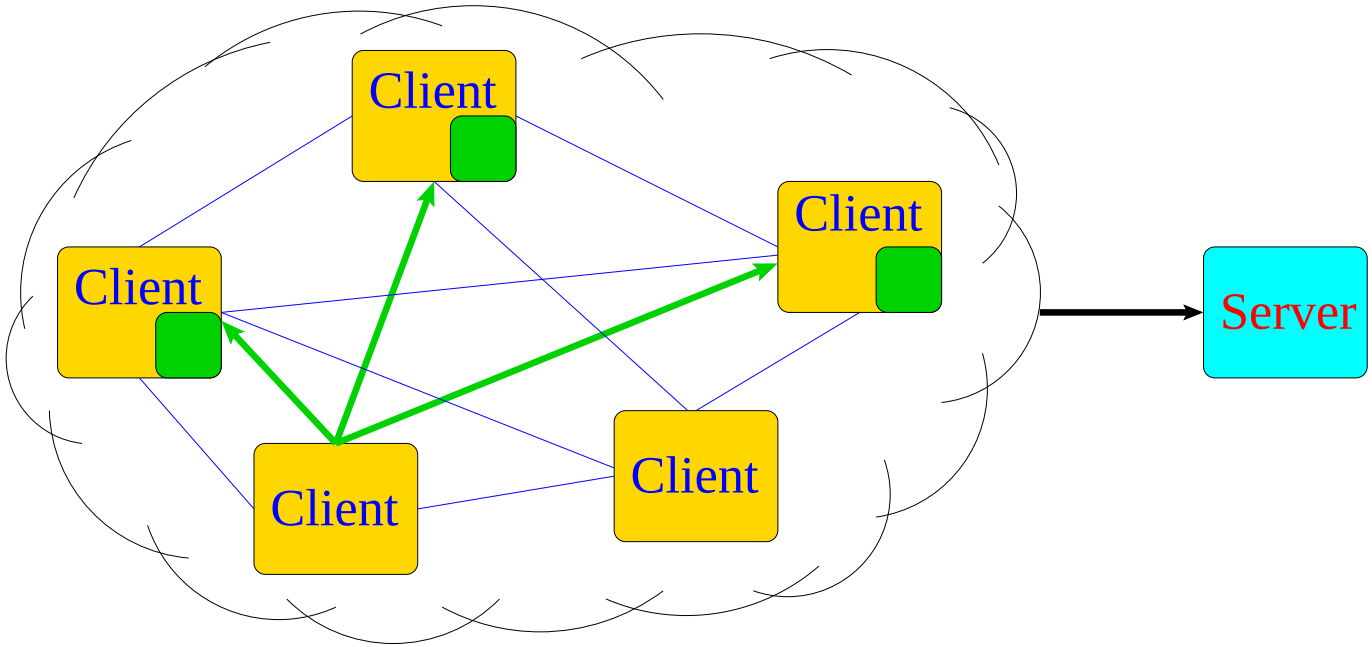
Clients fetch data from each other and offload burden from server




- Shark clients maintain a distributed index

# Scalability – Cooperative Caching

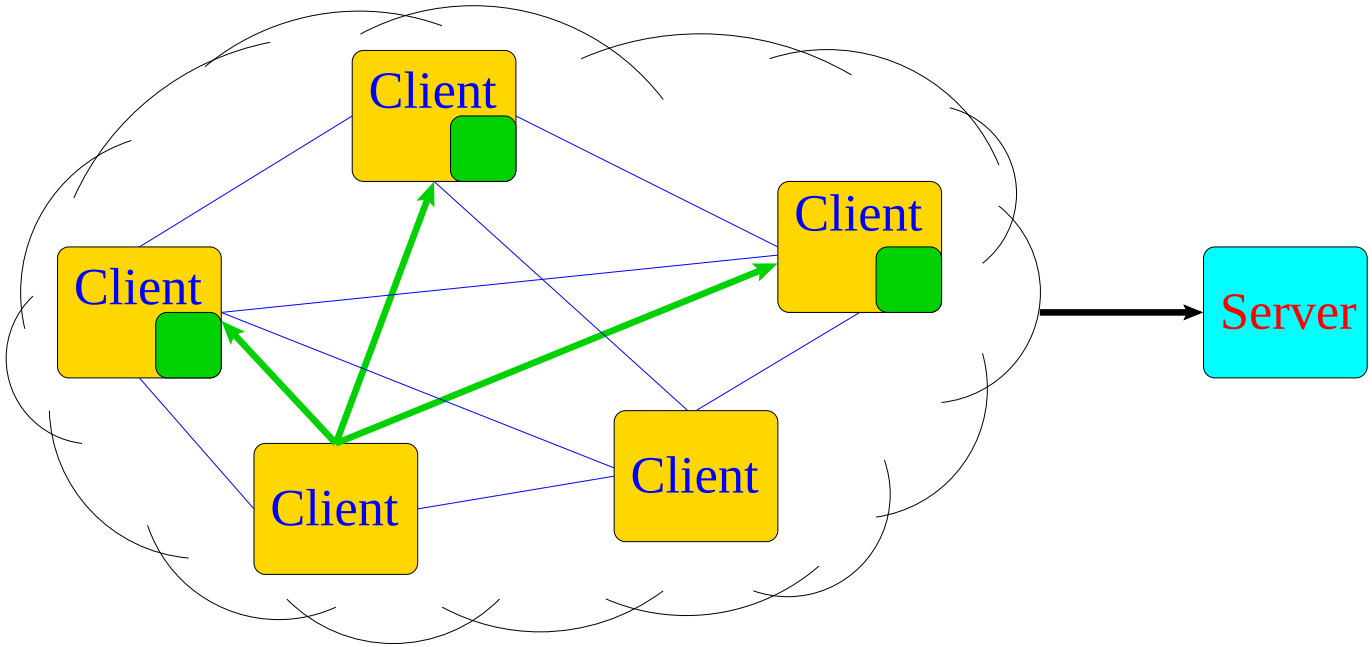
Clients fetch data from each other and offload burden from server




- Shark clients maintain a distributed index
- Fetch a file from multiple other clients in parallel

# Scalability – Cooperative Caching

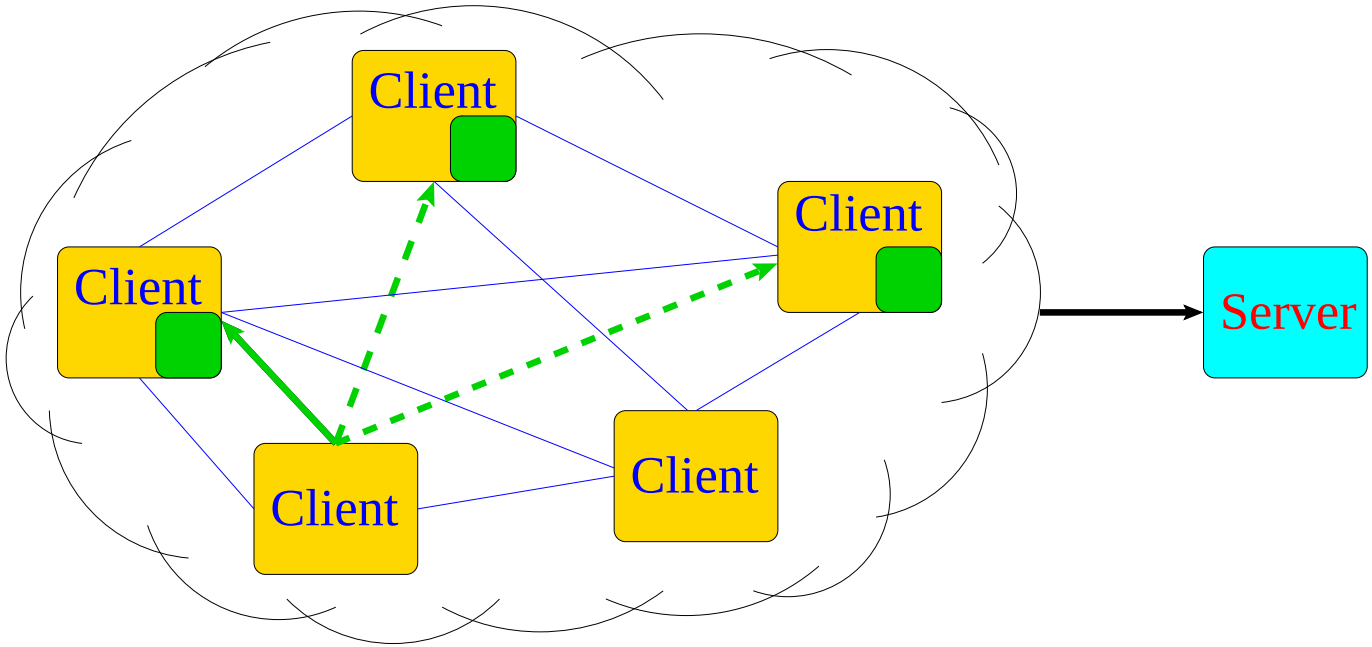
Clients fetch data from each other and offload burden from server




- Shark clients maintain a distributed index
- Fetch a file from multiple other clients in parallel
  - LBFS-style Chunks – Variable-sized blocks
  - Chunks are better – Exploit commonalities across files
  - Chunks preserved across file versions, concatenations

# Scalability – Cooperative Caching

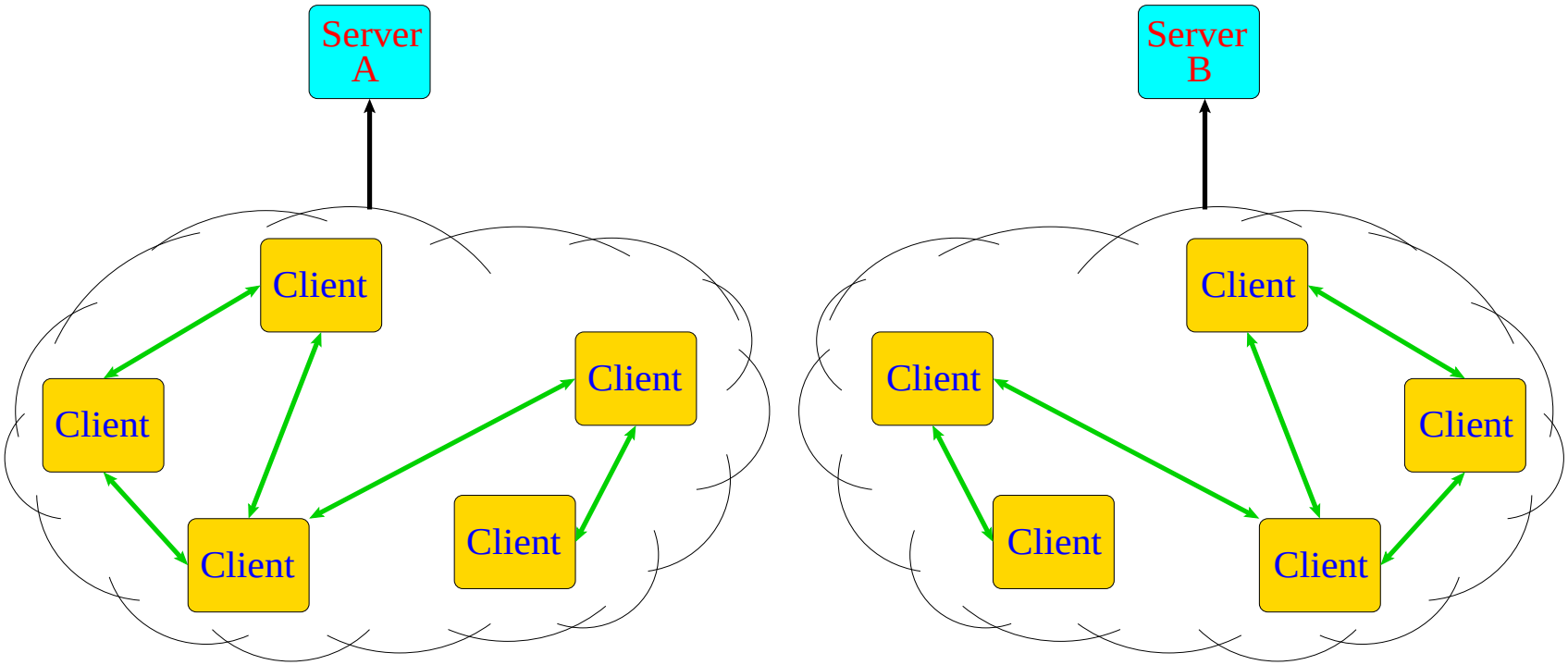
Clients fetch data from each other and offload burden from server




- Shark clients maintain a distributed index
- Fetch a file from multiple other clients in parallel
- Locality-awareness – Preferentially fetch chunks from nearby clients

# Cross-filesystem Sharing

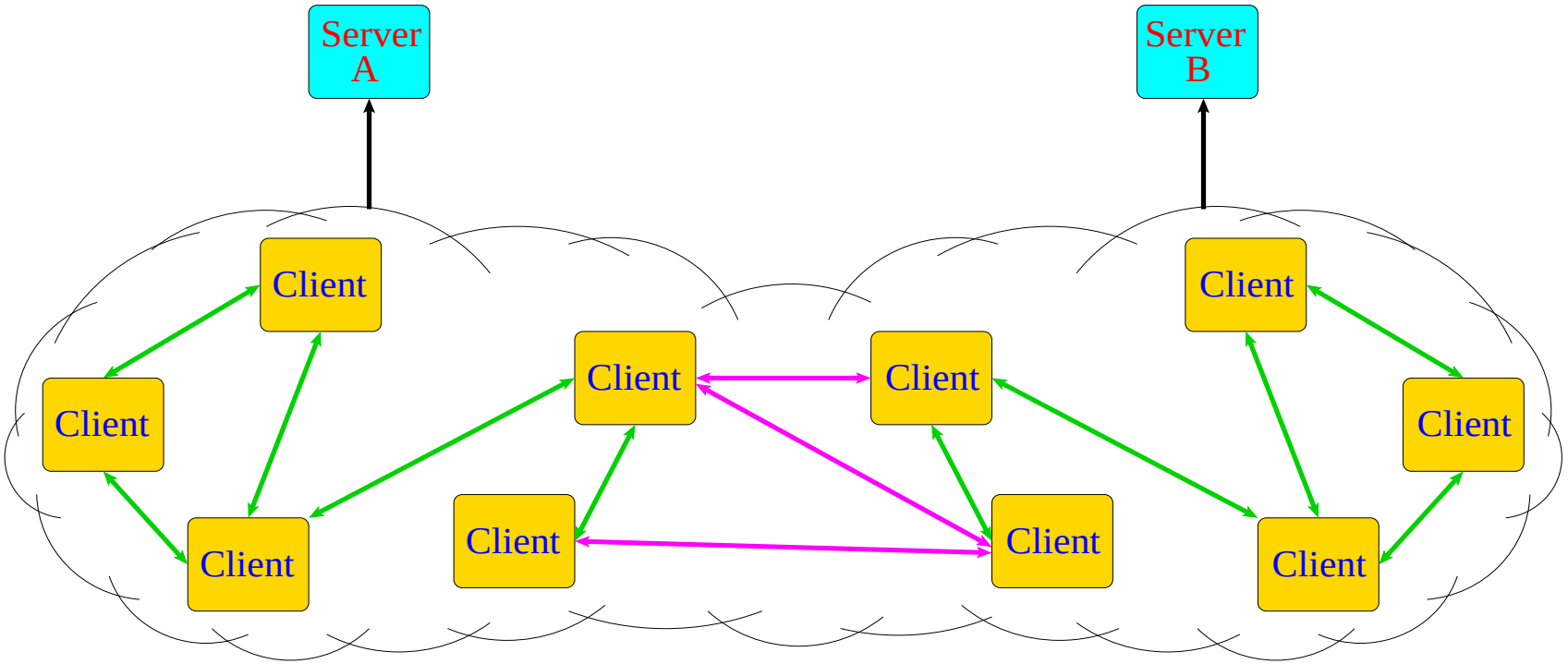
Global cooperative cache regardless of origin servers




- Two groups of clients accessing servers A and B
- Client groups share a large amount of software
- Such clients automatically form a global cache

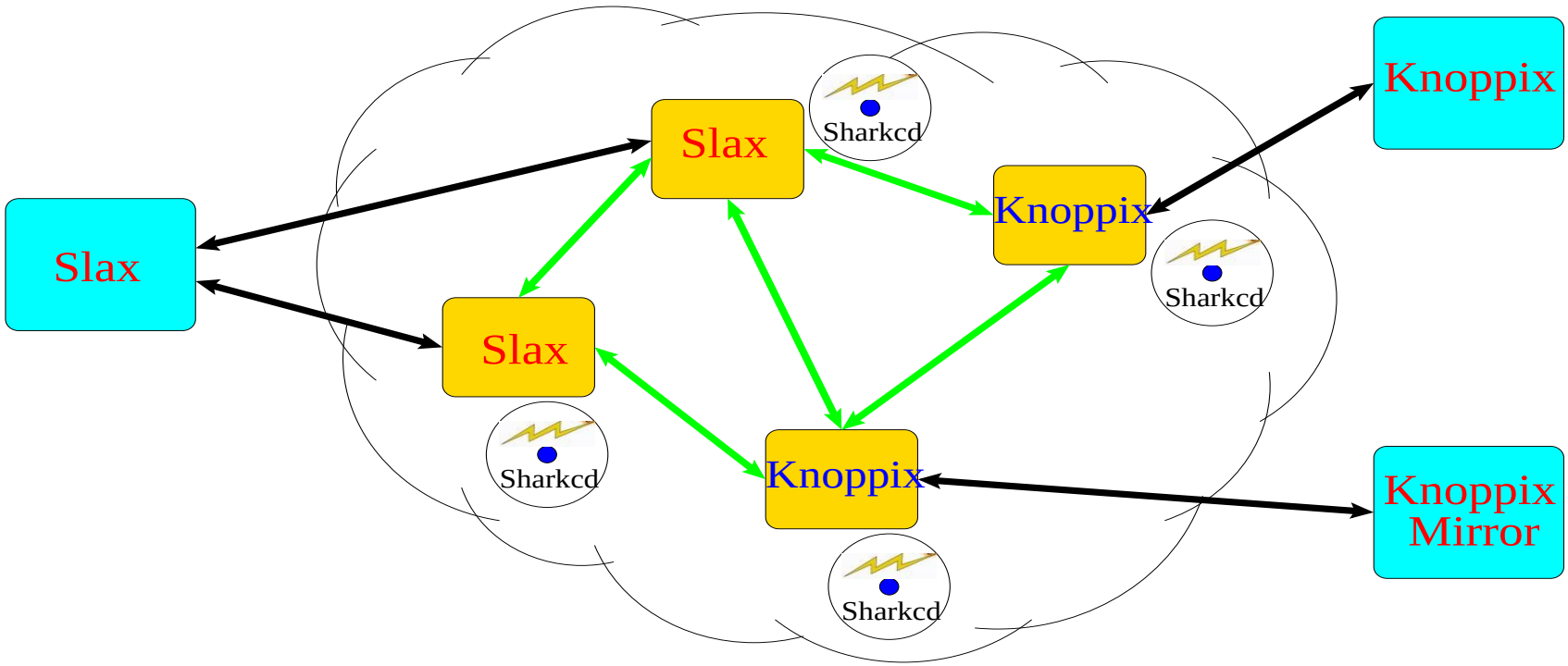
# Cross-filesystem Sharing

Global cooperative cache regardless of origin servers



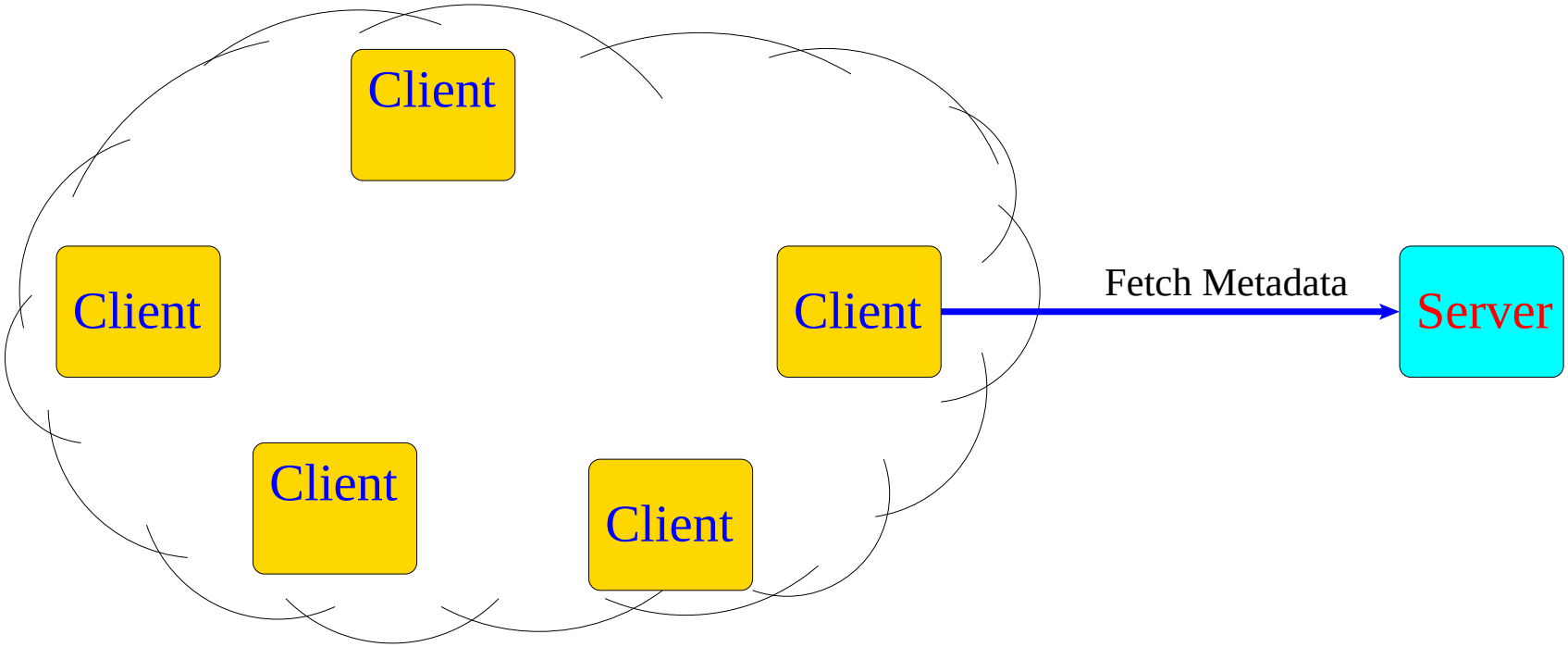

- Two groups of clients accessing servers A and B
- Client groups share a large amount of software
- Such clients automatically form a global cache

# Application



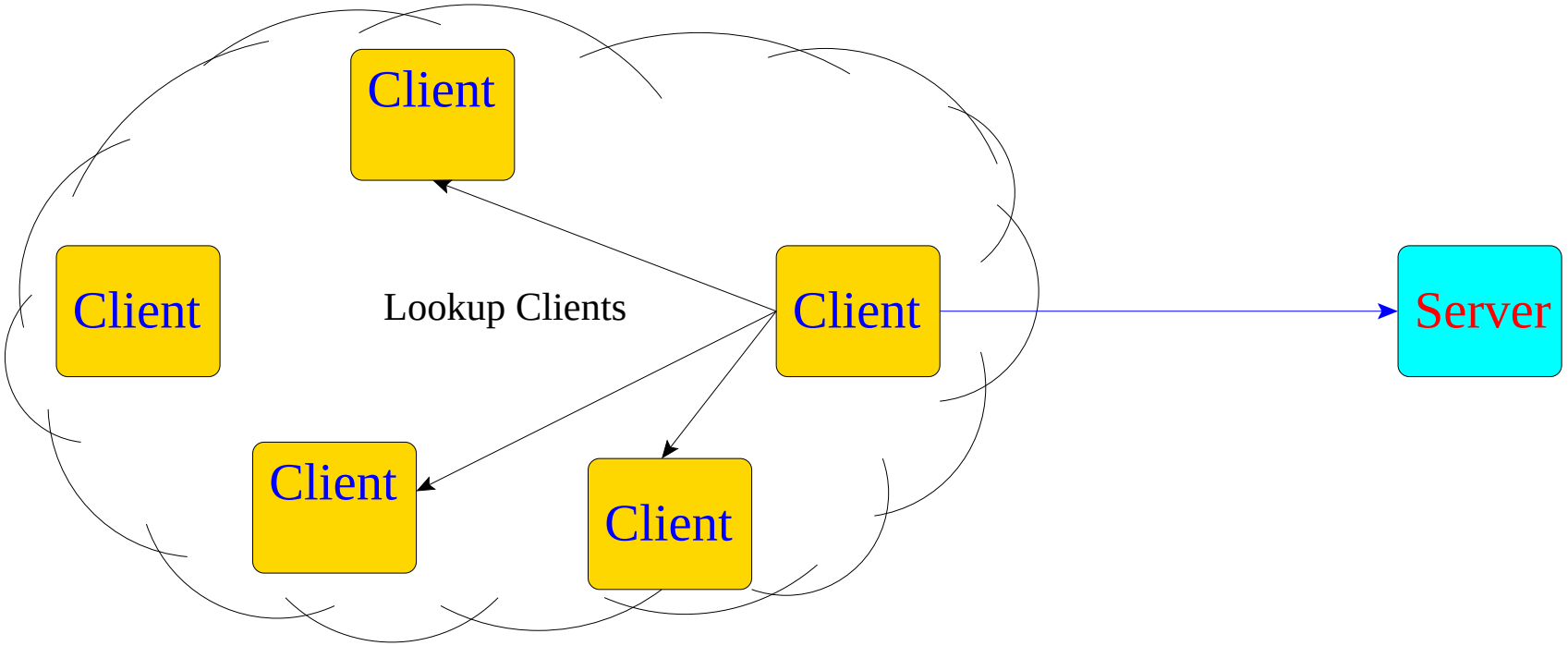

- LiveCD – Run an entire OS without using hard disk
- But all your programs must fit on a CD-ROM
- Download dynamically from server but scalability problems
- Knoppix and Slax users form global cache – Relieve servers

# Cooperative Caching – Roadmap



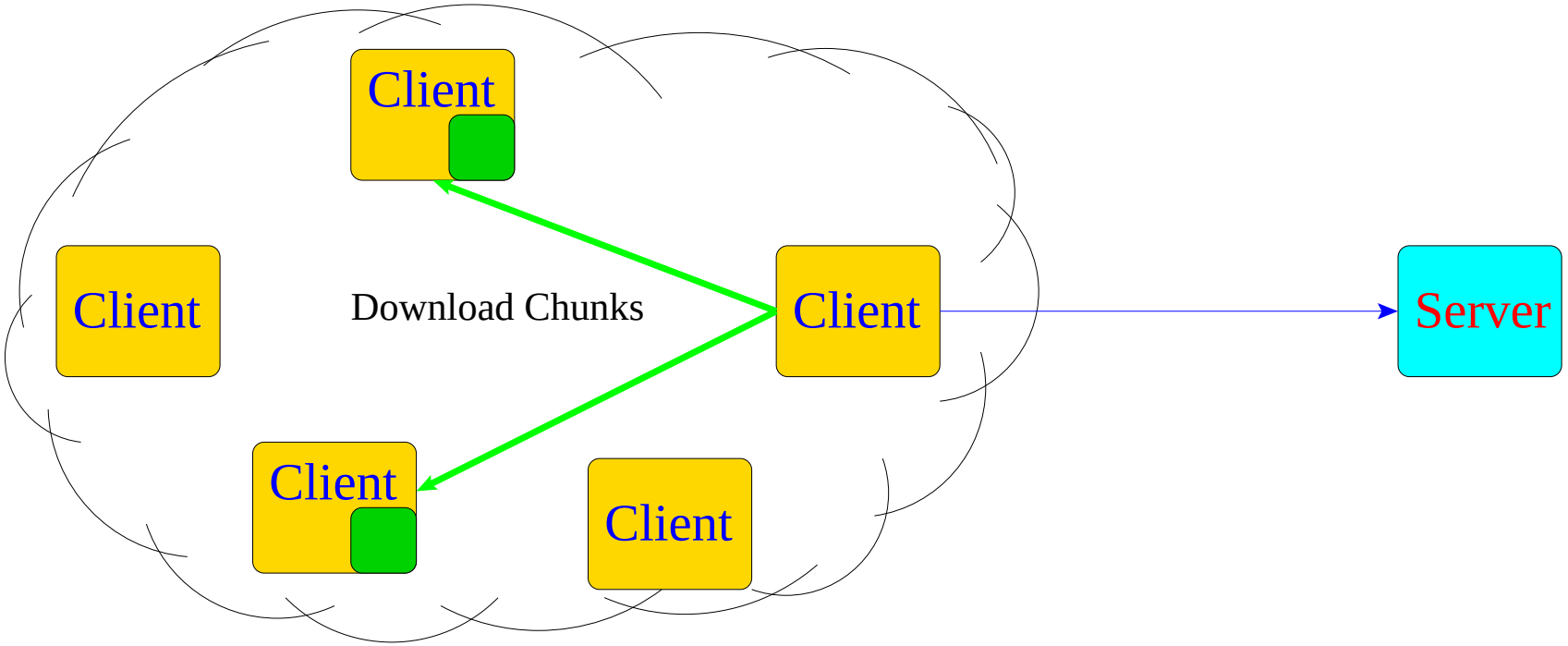

- Fetch metadata from the server

# Cooperative Caching – Roadmap



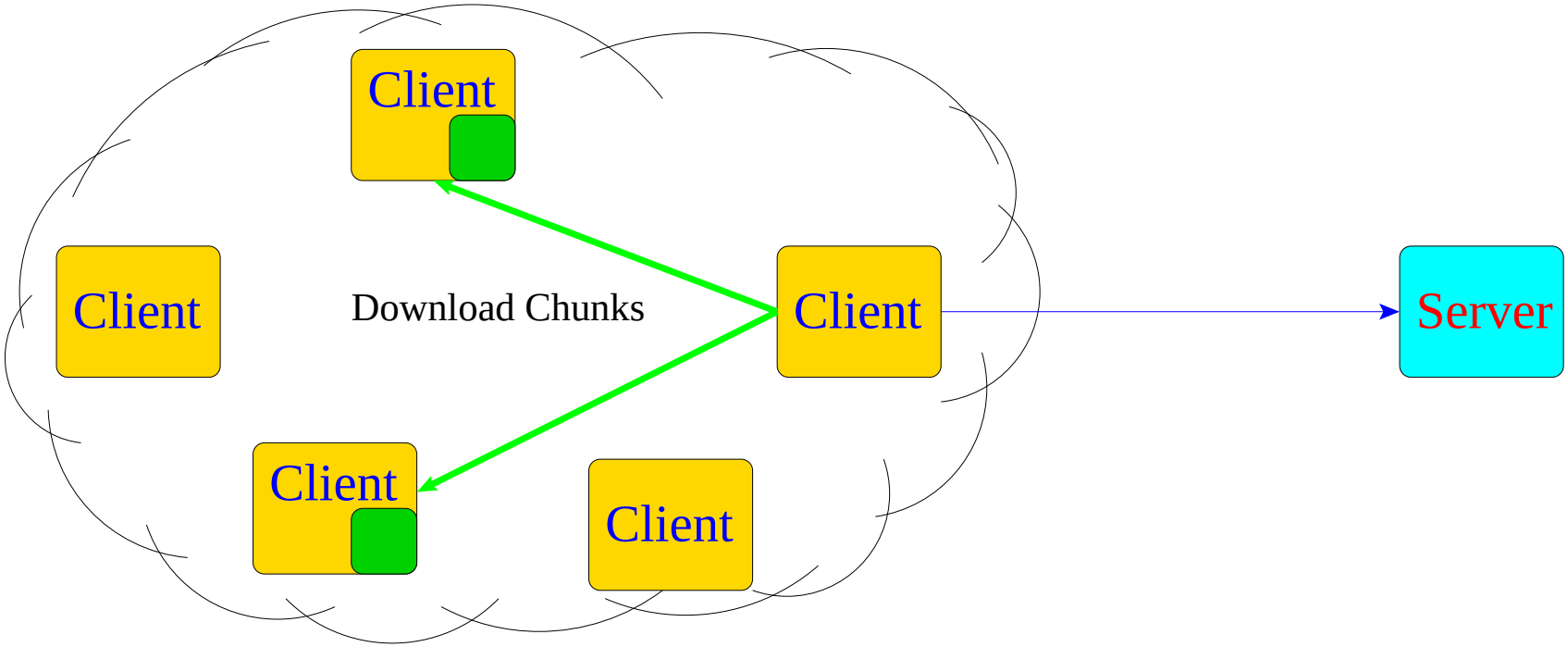

- Fetch metadata from the server
- Look up clients caching needed chunks in overlay

# Cooperative Caching – Roadmap




- Fetch metadata from the server
- Look up clients caching needed chunks in overlay
- Connect to multiple clients and download chunks in parallel

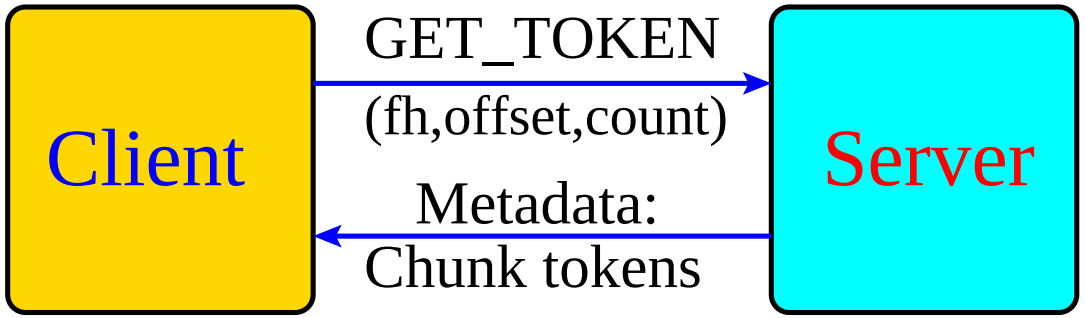
# Cooperative Caching – Roadmap



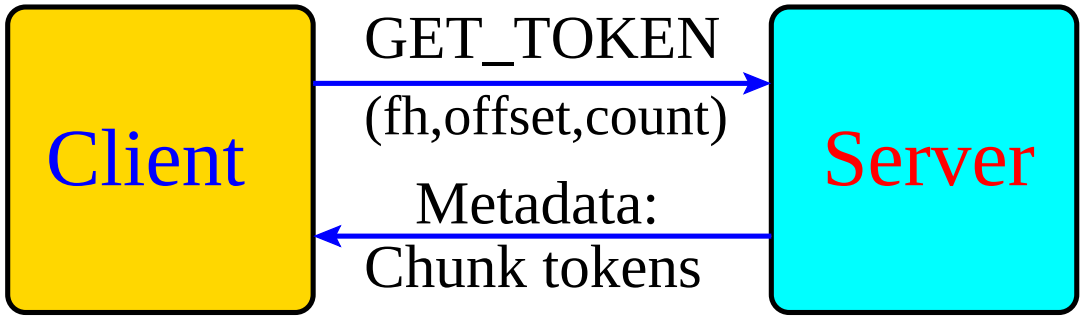

- Fetch metadata from the server
- Look up clients caching needed chunks in overlay
- Connect to multiple clients and download chunks in parallel
- Check integrity of fetched chunks

Clients are mutually distrustful

# File Metadata

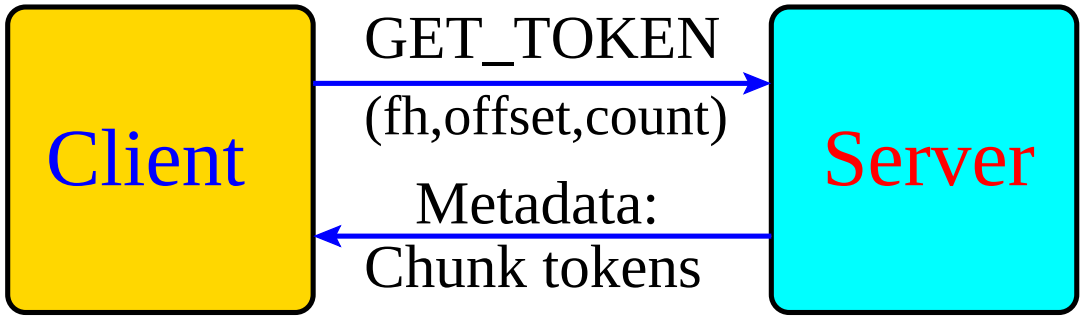


# File Metadata



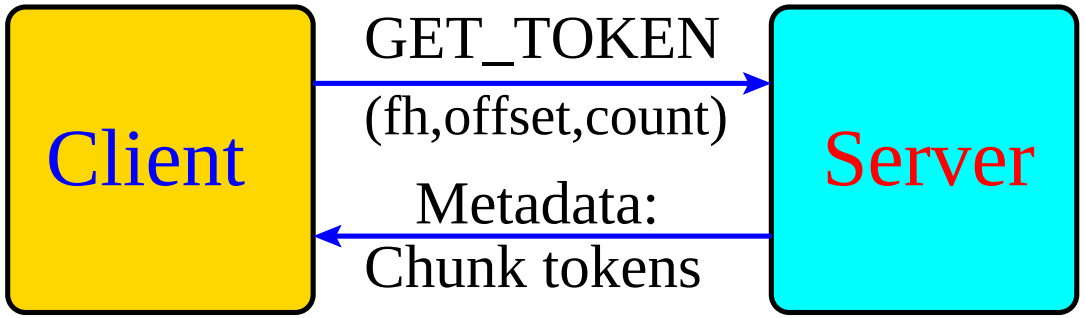

- **Possession of token implies server permissions to read**

# File Metadata




- **Possession of token implies server permissions to read**
- **Tokens are a shared secret between authorized clients**

# File Metadata

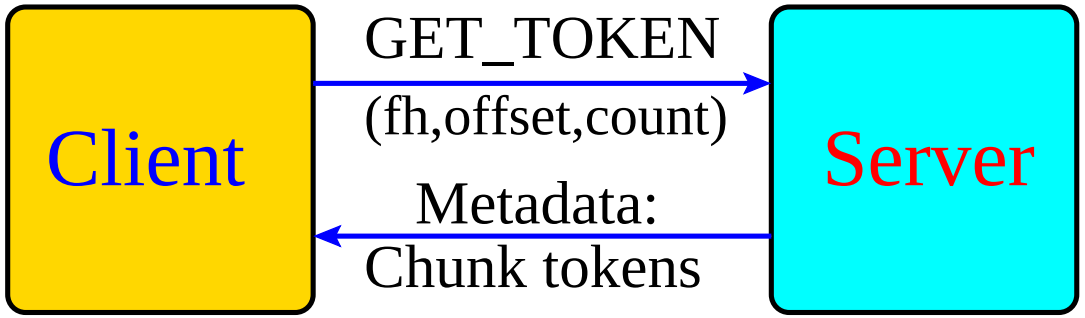



- **Possession of token implies server permissions to read**
- **Tokens are a shared secret between authorized clients**

Given a chunk B...

- Chunk token $T_B = H(B)$
- H is a collision resistant hash function

# File Metadata

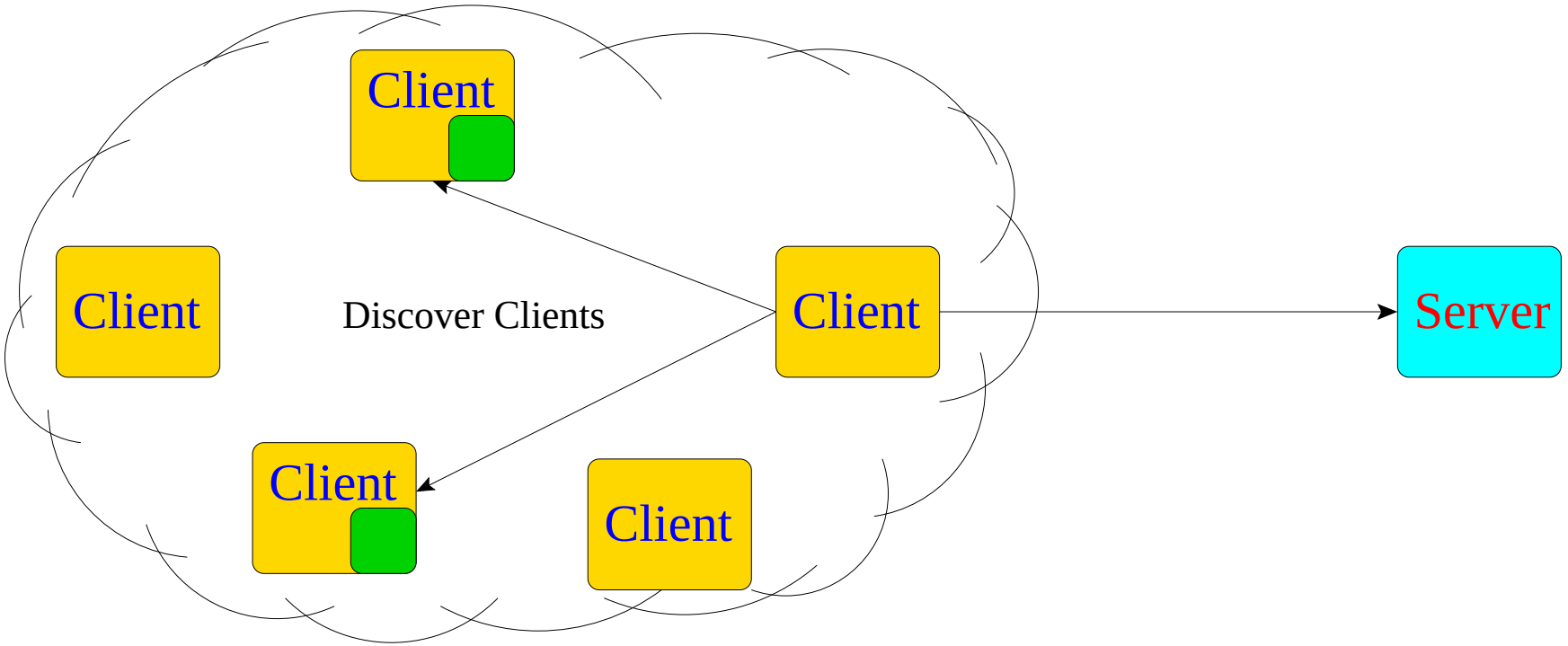



- **Possession of token implies server permissions to read**
- **Tokens are a shared secret between authorized clients**
- **Tokens can be used to check integrity of fetched data**

Given a chunk B...

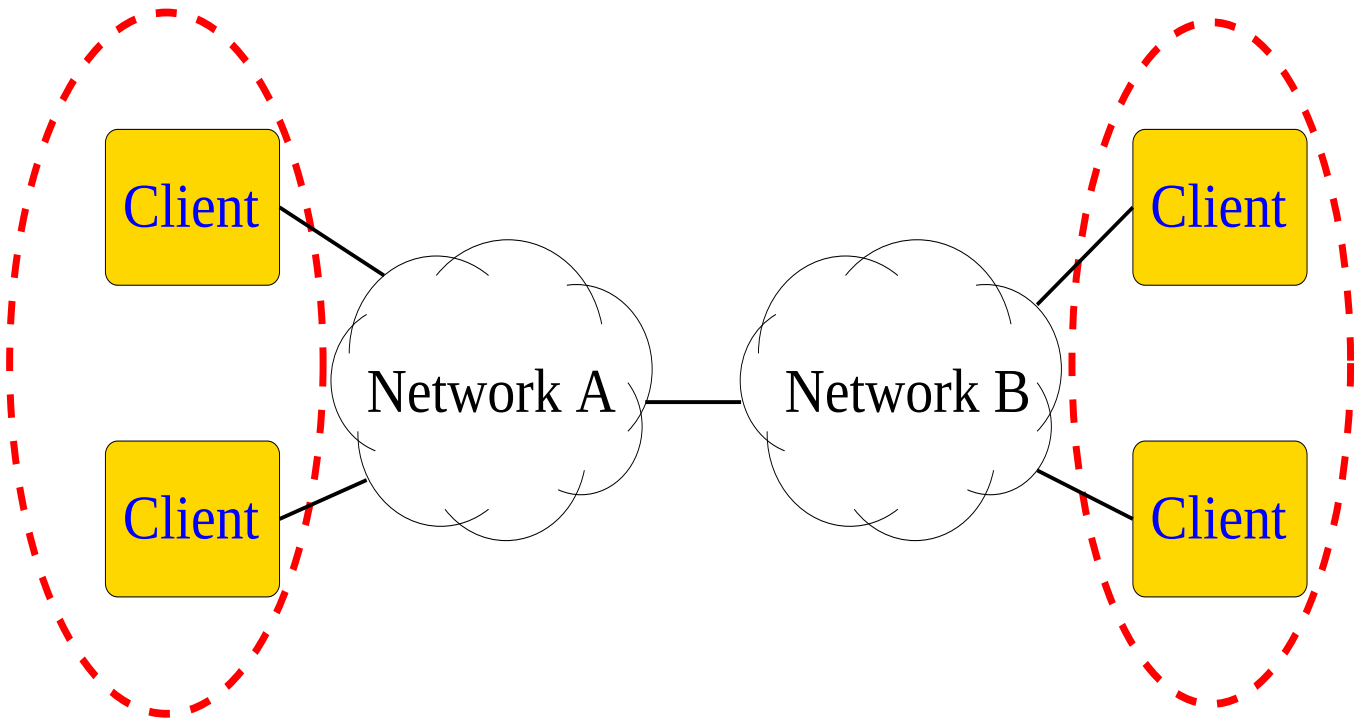
- Chunk token $T_B = H(B)$
- H is a collision resistant hash function

# Discovering Clients Caching Chunks




- For every chunk B, there's indexing key $I_B$
- $I_B$ used to index clients caching B
- Cannot set $I_B = T_B$, as $T_B$ is a secret
  - $I_B$ = MAC ($\mathbf{T_B}$, "Indexing Key")

# Locality Awareness




- Overlay organized as clusters based on latency
- Indexing infrastructure preferentially returns sources in same cluster as the client
- Hence, chunks usually transferred from nearby clients

# Final Steps

**Download Chunk**

- Security issues discussed later

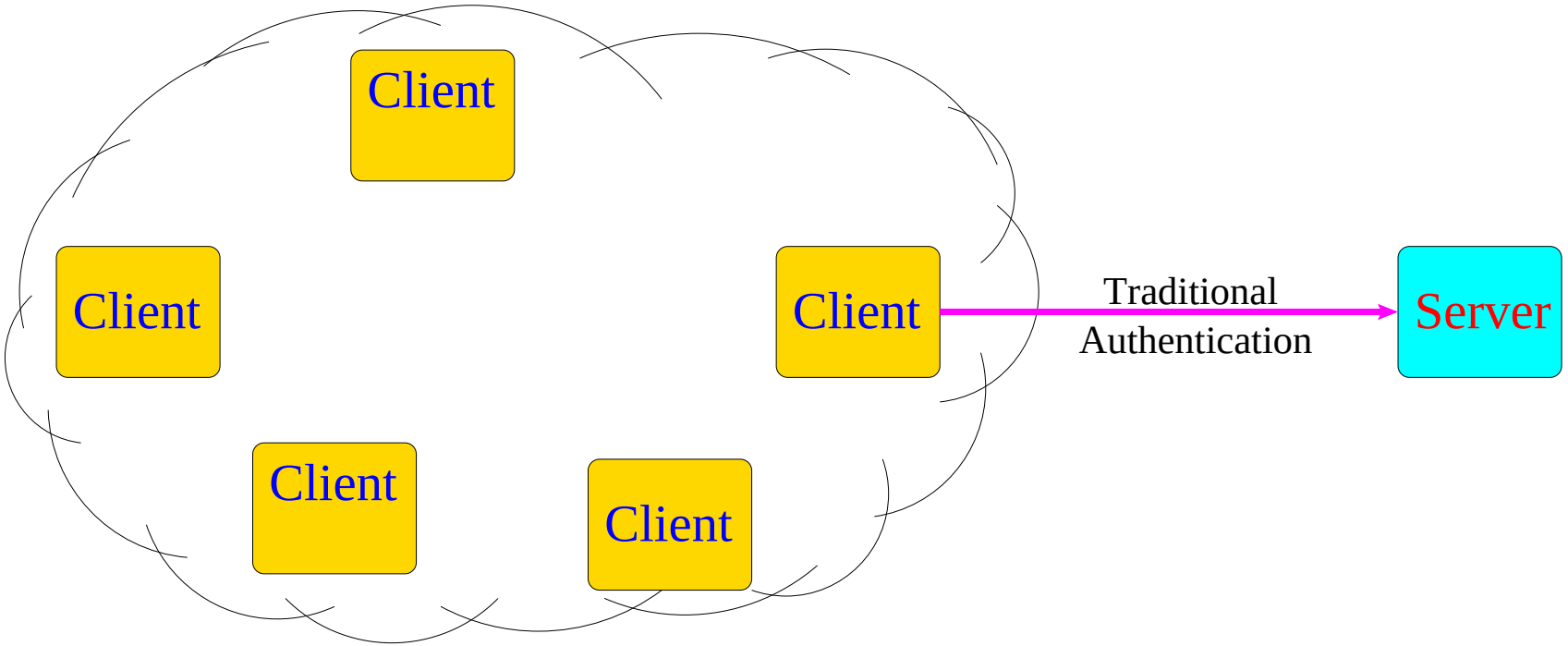
**Register as a source**

- Client now becomes a source for the downloaded chunk
- Client registers in distributed index – PUT($I_B$, Addr)

**Chunk Reconciliation**

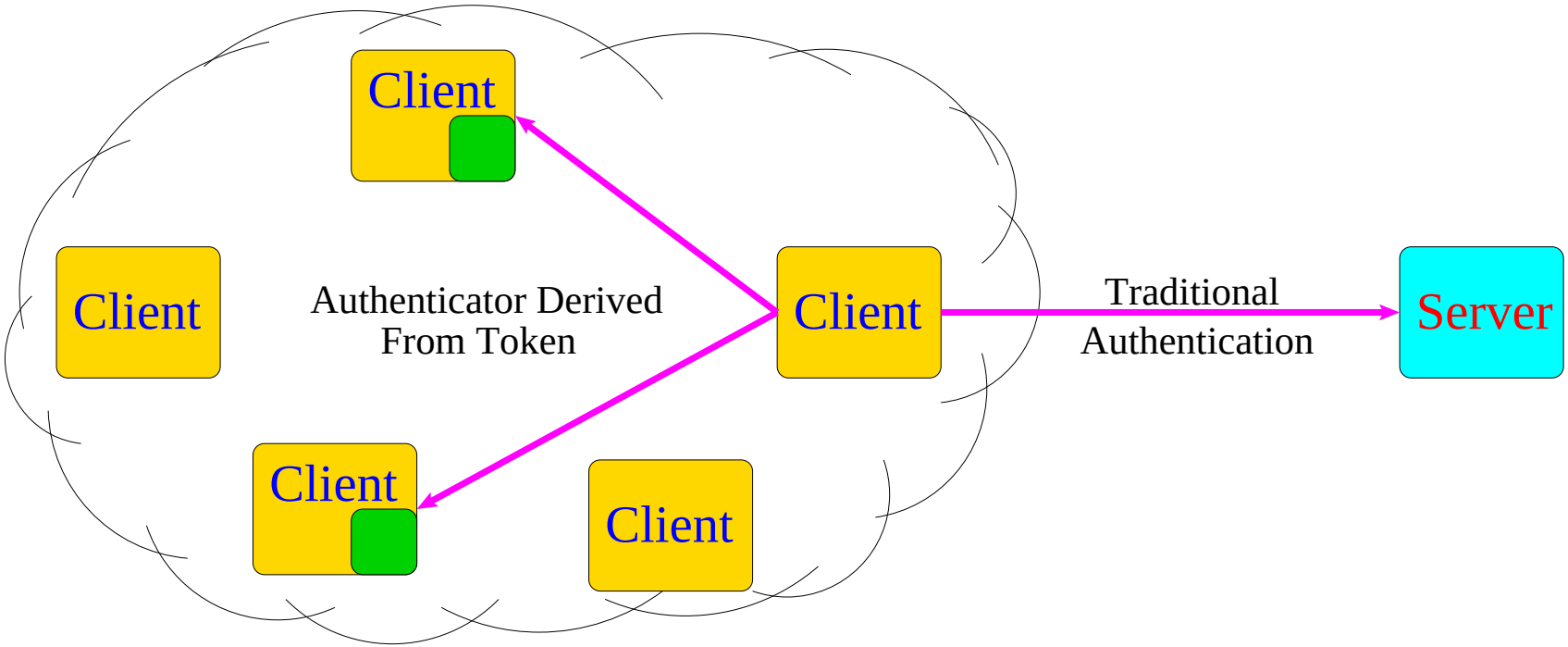
- Reuse connection to download more chunks
- Exchange mutually needed chunks w/o indexing overhead

# Security Issues – Client Authentication




- Traditionally, server authenticated read requests using `uids`

# Security Issues – Client Authentication

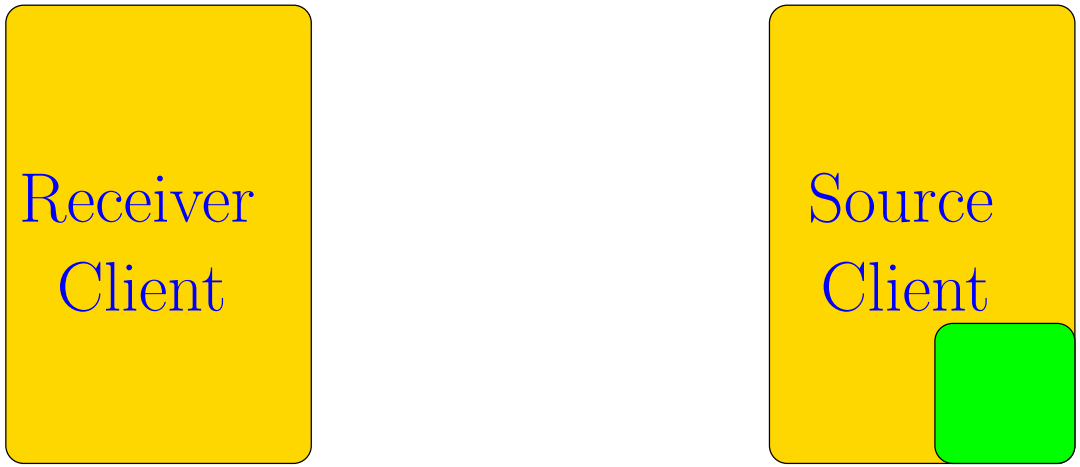



- Traditionally, server authenticated read requests using `uids`
- Challenge – How does a client know when to send chunks?
- Chunk token allows client to identify authorized clients

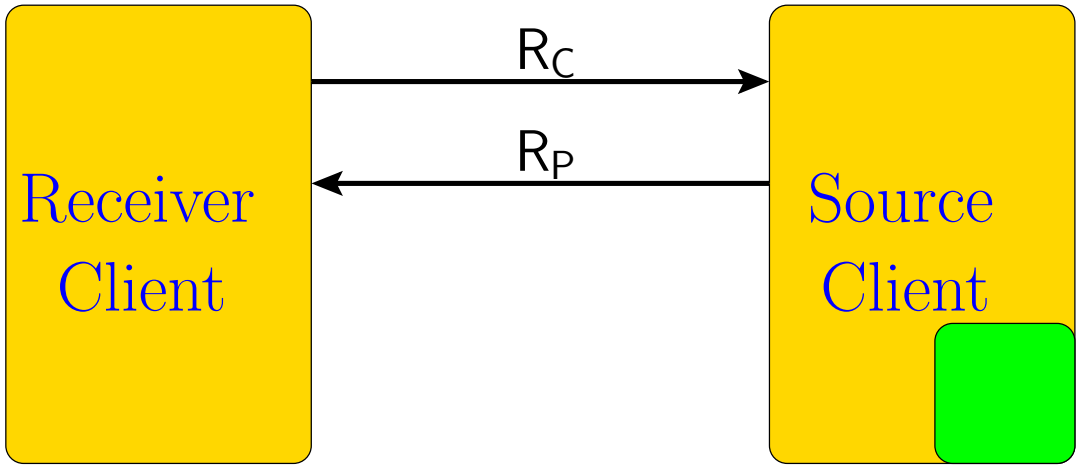
# Security Issues – Client Communication

- Client should be able to check integrity of downloaded chunk
- Client should not send chunks to other unauthorized clients
- An eavesdropper shouldn't be able to obtain chunk contents

# Security Protocol

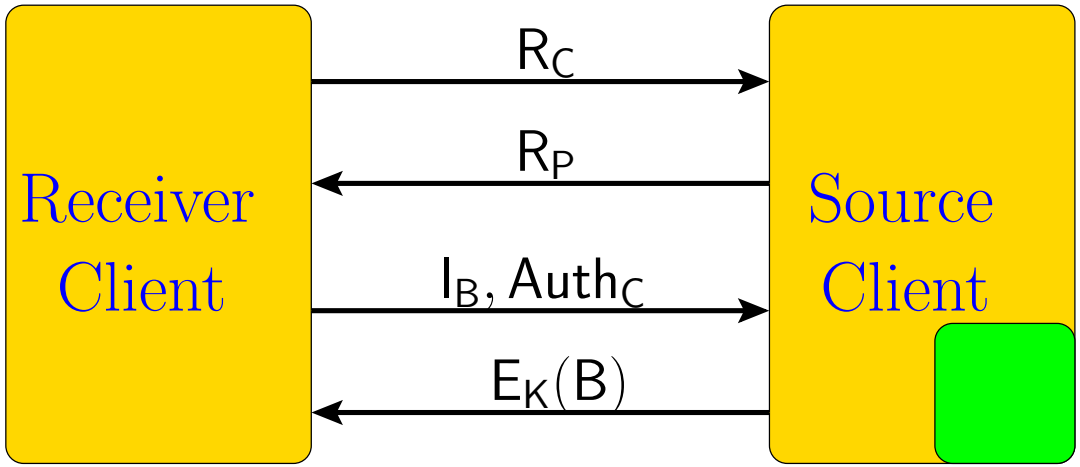


# Security Protocol



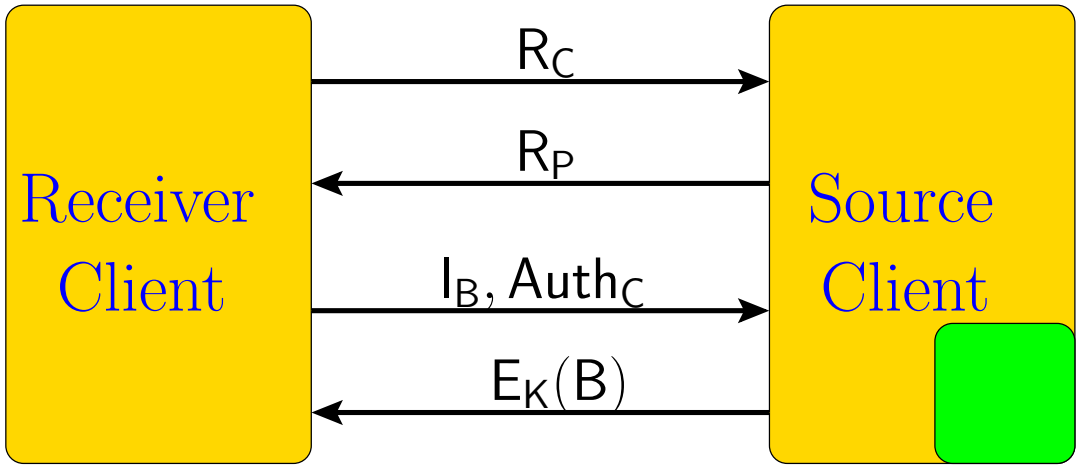

- $R_C, R_P$ – Random nonces to ensure freshness

# Security Protocol



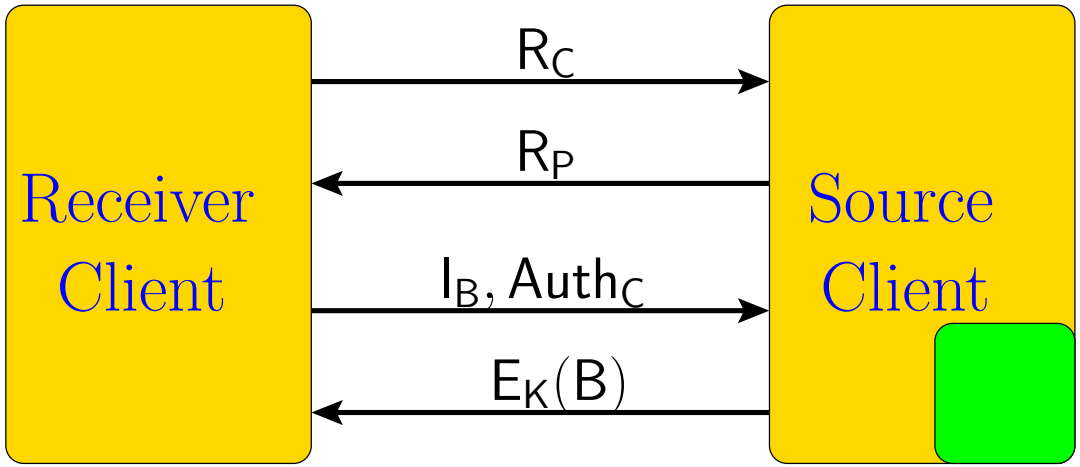

- $R_C, R_P$ – Random nonces to ensure freshness
- $Auth_C$ – Authenticator to prove receiver has token
  - $Auth_C = MAC\ (\mathbf{T_B},\ \text{“Auth C”},\ C, P, R_C, R_P)$

# Security Protocol




- $R_C, R_P$ – Random nonces to ensure freshness
- $Auth_C$ – Authenticator to prove receiver has token
  - $Auth_C = MAC\ (\mathbf{T_B},\ \text{“Auth C”},\ C, P, R_C, R_P)$
- K – Key to encrypt chunk contents
  - $K = MAC\ (\mathbf{T_B},\ \text{“Encryption”},\ C, P, R_C, R_P)$

# Security Properties




**Client can check integrity of downloaded chunk**

- Client checks H(Downloaded chunk) $\stackrel{?}{=} T_B$

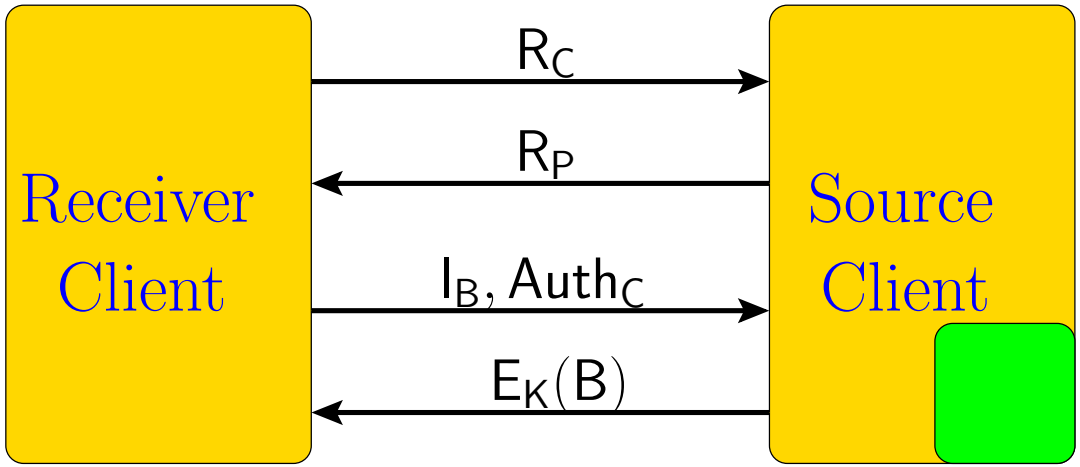
**Source should not send chunks to unauthorized clients**

- Malicious clients cannot send correct $Auth_C$

**Eavesdropper shouldn't get chunk contents**

- All communication encrypted with K

# Security Properties




Privacy limitations for world-readable files

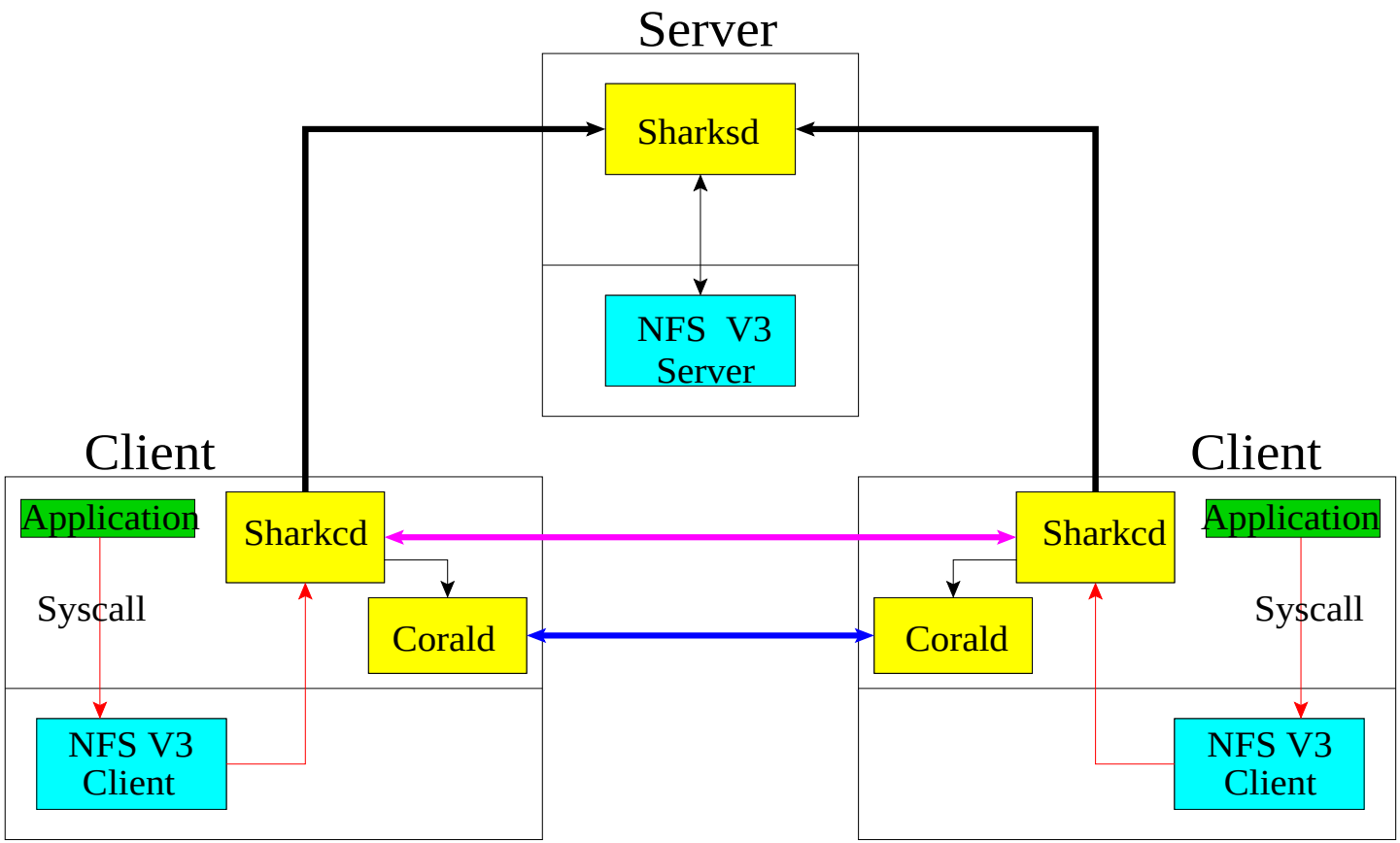
- Eavesdropper can track lookups of clients
- Eavesdropper hashes data, finds what exactly client downloads

For private files, solution described in paper

- Sacrifices cross-FS sharing for better privacy

Forward Secrecy not guaranteed

# Implementation

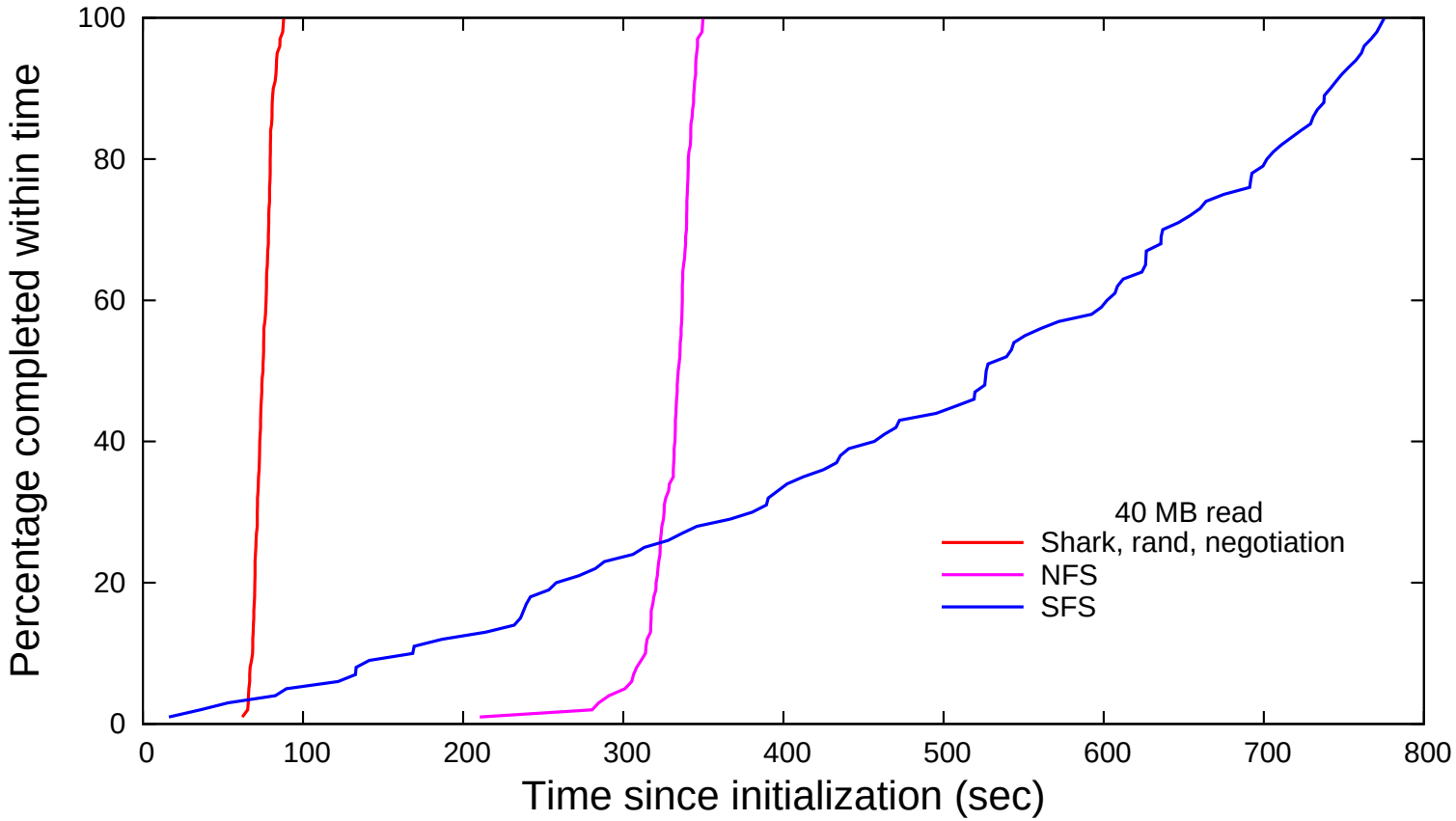



- `sharkcd` – Incorporates source-receiver client functionality
- `sharksd` – Incorporates chunking mechanisms
- `corald` – A node in the indexing infrastructure

# Evaluation

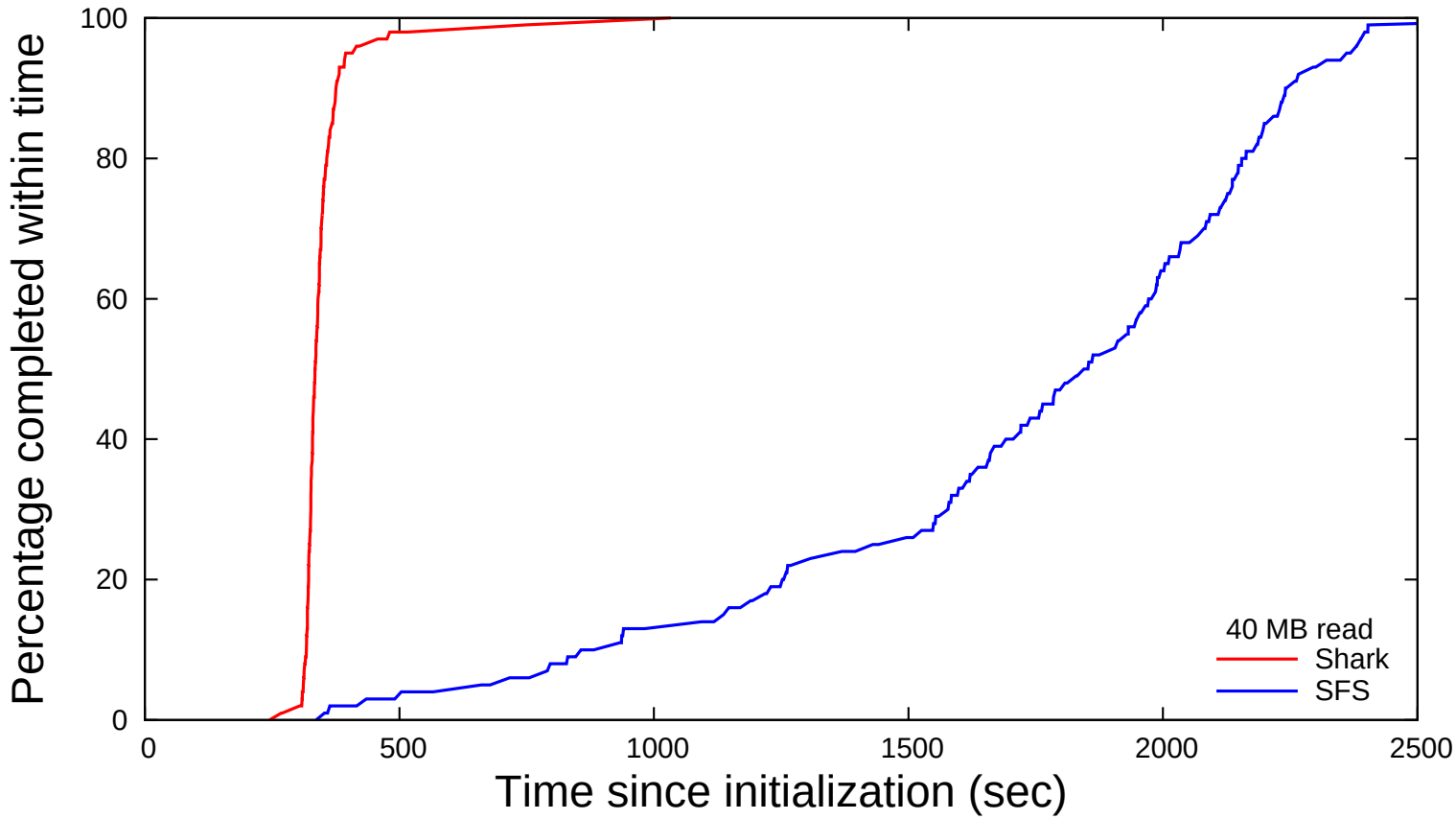
- How does Shark compare with SFS? With NFS?
- How scalable is the server?
- How fair is Shark across clients?
- Which order is better? Random or Sequential
- What are the benefits of set reconciliation?

# Emulab – 100 Nodes on LAN



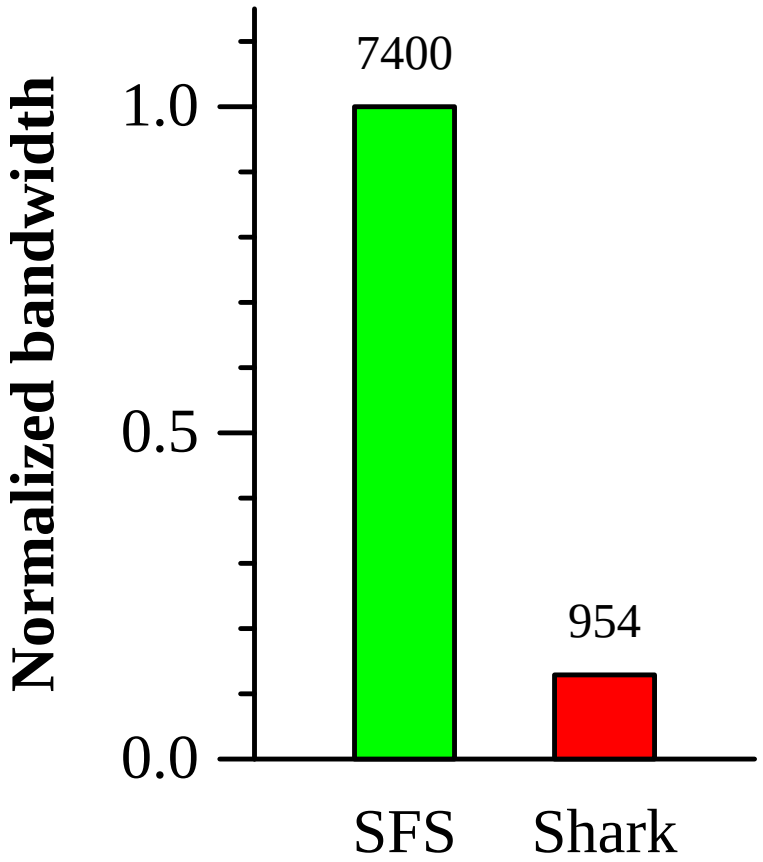

- Shark – 88s
- SFS – 775s ($\approx$ 9x better), NFS – 350s ($\approx$ 4x better)
- SFS less fair because of TCP backoffs

# PlanetLab – 185 Nodes



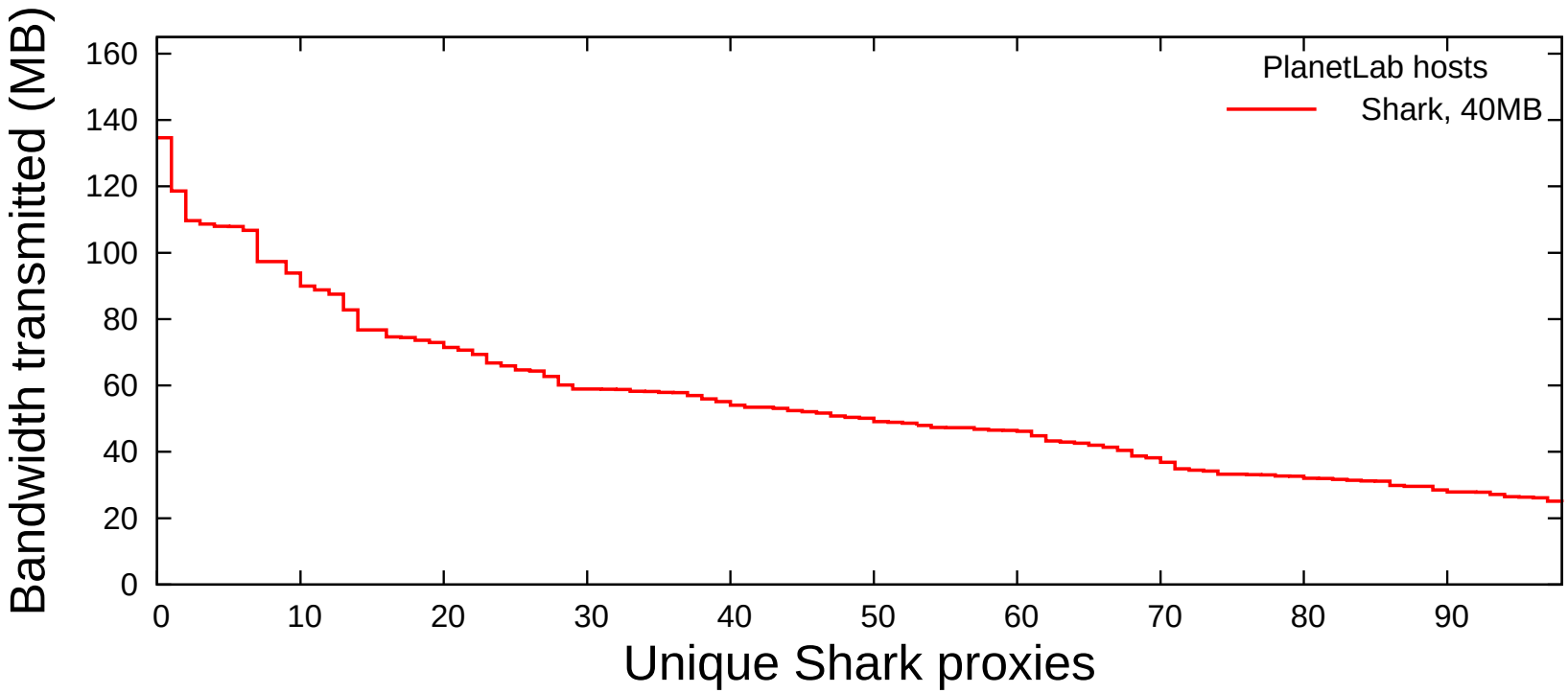

- Shark $\approx$ 7 min – 95th Percentile
- SFS $\approx$ 39 min – 95th Percentile (5x better)
- NFS – Triggered kernel panics in server

# Data pushed by Server




- Shark vs SFS – 23 copies vs 185 copies (8x better)

# Data served by Clients




- Maximum contribution $\approx$ 3.5 copies
- Median contribution $\approx$ 1.5 copies
- Minimum contribution $\approx$ 0.75 copies

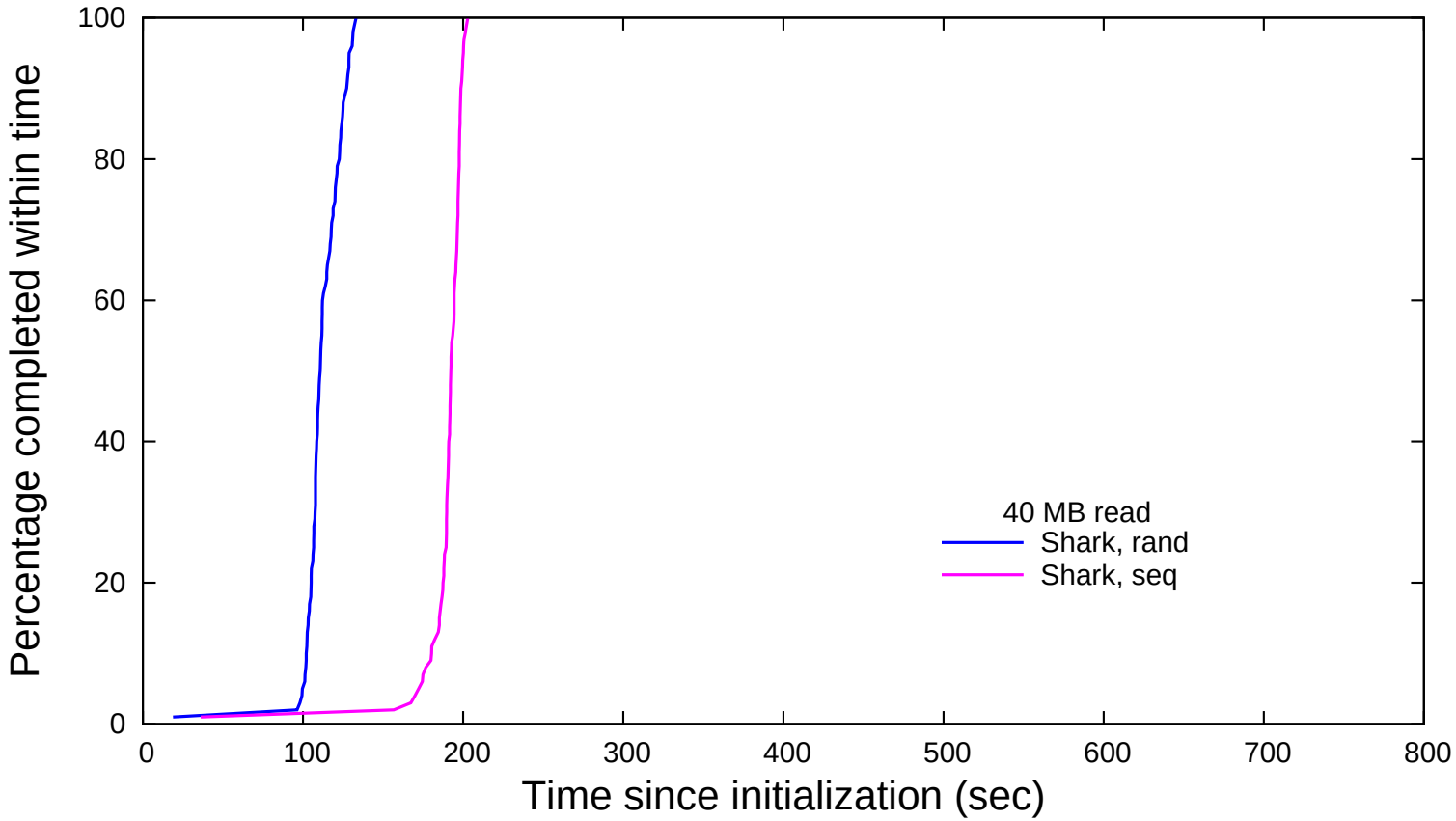
# Fetching Chunks – Order Matters

- In what order should we fetch chunks of a file?
- Natural choices – Random or Sequential

**Intuitively, when many clients start simultaneously**

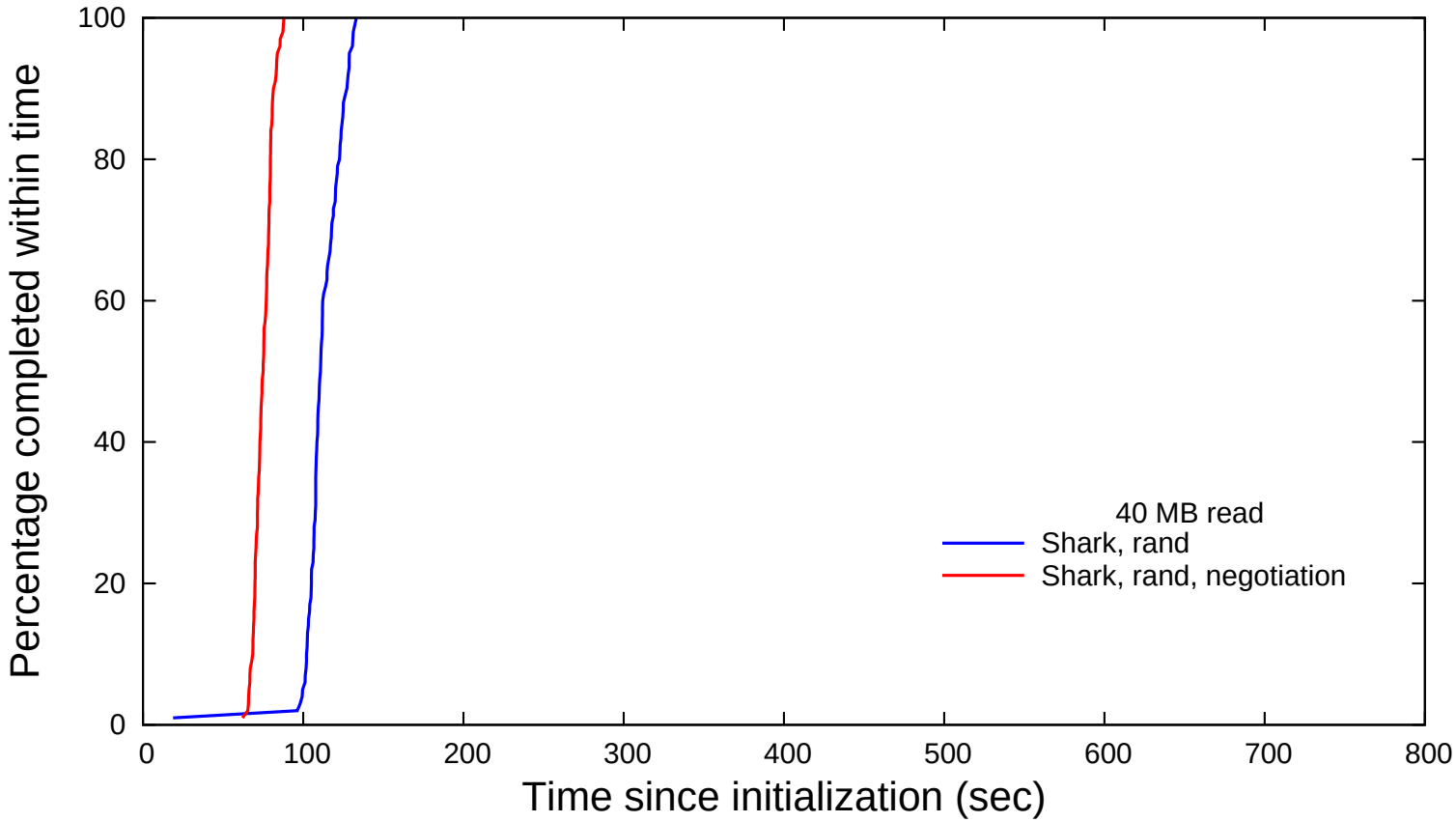
- Random
  - All clients fetch independent chunks
  - More chunks become available in the cooperative cache
- Sequential
  - Better disk I/O scheduling on the server
  - Client that downloads most chunks alone adds to cache

# Emulab – 100 Nodes on LAN




- Random – 133s
- Sequential – 203s
- Random Wins !!! – 35% better

# Emulab – 100 Nodes on LAN




- Random + Reconciliation – 88s
- Random – 133s
- Reconciliation crucial – 34% improvement

# Conclusions

- Networked filesystems offer a convenient interface
- Current networked filesystems like NFS are not scalable
  - Forces users to resort to inconvenient tools like rsync
- Shark offers a filesystem that scales to hundreds of clients
  - Locality-aware cooperative cache
  - Supports cross-FS sharing enabling novel applications

# Questions

http://www.scs.cs.nyu.edu/shark