Scaling Data Servers via Cooperative Caching

Siddhartha Annapureddy

New York University

Committee:
David Mazières, Lakshminarayanan Subramanian, Jinyang Li, Dennis Shasha, Zvi Kedem
Motivation

New functionality over the Internet

- Large number of users consume many (large) files
  - YouTube, Azureus etc. for movies – DVDs no more
  - Data dissemination to large distributed farms
Current approaches

- Scalability
  - Server's uplink saturates
  - Operation takes long time to finish

Siddhartha Annapureddy
Thesis Defense
Current approaches

- **Scalability** ↓ – Server’s uplink saturates
  - Operation takes long time to finish
Current approaches (contd.)

- Server farms replace central server
  - Scalability problem persists
  - Notoriously expensive to maintain

- Content Distribution Networks (CDNs)
  - Still quite expensive

- Peer-to-peer (P2P) systems
  - A new development
Peer-to-peer (P2P) systems

- Low-end central server
- Large client population
- File usually divided into chunks/block
- Use client resources to disseminate file
  - Bandwidth, disk space etc.
- Clients organized into a connected overlay
  - Each client knows only a few other clients

How to design P2P systems to scale data servers?
In order to achieve scalability, need to avoid hotspots

- **Keep bw load on server to a minimum**
  - Server gives data only to few users
    - Leverage participants to distribute to other users
    - Maintenance overhead with users to be kept low
- **Keep bw load on users to a minimum**
  - Each node interacts with only a small subset of users
P2P systems – Themes

- Usability – Exploiting P2P for a range of systems
  - File systems, user-level apps etc.
- Security issues in such wide-area P2P systems
  - Providing data integrity, privacy, authentication
- Cross file(system) sharing – Use other groups
  - Make use of all available sources
- Topology management – Arrange nodes efficiently
  - Underlying network (locality properties)
  - Node capacities (bandwidth, disk space etc.)
  - Application-specific (certain nodes together work well)
- Scheduling dissemination of chunks efficiently
  - Nodes only have partial information
P2P systems – Themes

- **Usability – Exploiting P2P for a range of systems**
  - File systems, user-level apps etc.
- **Security issues in such wide-area P2P systems**
  - Providing data integrity, privacy, authentication
- **Cross file(system) sharing – Use other groups**
  - Make use of all available sources
- **Topology management – Arrange nodes efficiently**
  - Underlying network (locality properties)
  - Node capacities (bandwidth, disk space etc.)
  - Application-specific (certain nodes together work well)
- **Scheduling dissemination of chunks efficiently**
  - Nodes only have partial information
P2P systems – Themes

- **Usability** – Exploiting P2P for a range of systems
  - File systems, user-level apps etc.
- **Security issues** in such wide-area P2P systems
  - Providing data integrity, privacy, authentication
- **Cross file/system** sharing – Use other groups
  - Make use of all available sources
- **Topology management** – Arrange nodes efficiently
  - Underlying network (locality properties)
  - Node capacities (bandwidth, disk space etc.)
  - Application-specific (certain nodes together work well)
- **Scheduling dissemination** of chunks efficiently
  - Nodes only have partial information
P2P systems – Themes

- **Usability – Exploiting P2P for a range of systems**
  - File systems, user-level apps etc.

- **Security issues in such wide-area P2P systems**
  - Providing data integrity, privacy, authentication

- **Cross file(system) sharing – Use other groups**
  - Make use of all available sources

- **Topology management – Arrange nodes efficiently**
  - Underlying network (locality properties)
  - Node capacities (bandwidth, disk space etc.)
  - Application-specific (certain nodes together work well)

- **Scheduling dissemination of chunks efficiently**
  - Nodes only have partial information
Shark – 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

Distribute from a central server

- rsync, unison, scp

- Server’s network uplink saturates
- Wastes bandwidth along bottleneck links
Current approaches

File distribution mechanisms

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

BitTorrent, Bullet
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 ⇒ Redistribution
- Need low cost solution for refetching files

Manual management of software versions
- Point /usr/local to a central server

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

![Graph showing scalability]

Time to finish Read (sec) vs. Unique nodes

- **Very slow**: ≈ 775s
- **Much better!!!**: ≈ 88s (9x better)

Siddhartha Annapureddy | Thesis Defense
Problem: Scalability

![Graph showing time to finish read (sec) vs. unique nodes for SFS and Shark]

- Very slow $\approx 775s$
- Much better !!! $\approx 88s$ (9x better)

Siddhartha Annapureddy | Thesis Defense
Let’s design this new filesystem

Vanilla SFS

- Central server model
Scalability – Client-side caching

Large client cache à la AFS

- Whole file caching
- Scales by avoiding redundant data transfers
- Leases to ensure NFS-style cache consistency
Scalability – Cooperative caching

- With multiple clients, must address bandwidth concerns
Scalability – Cooperative caching

Clients fetch data from each other and offload burden from server

- Shark clients maintain distributed index
Scalability – Cooperative caching

Clients fetch data from each other and offload burden from server

- Shark clients maintain 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 distributed index
- Fetch a file from multiple other clients in parallel
- Read-heavy workloads – Writes still go to server
P2P systems – Themes

- Usability – Exploiting P2P for a range of systems
- Cross file(system) sharing – Use other groups
- Security issues in such wide-area P2P systems
- Topology management – Arrange nodes efficiently
- Scheduling dissemination of chunks efficiently
Cross file(system) sharing – Application

Linux distribution with LiveCDs

- 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
Cross file(system) sharing

Global cooperative cache regardless of origin servers

- Two groups of clients accessing Slax/Knoppix servers
- Client groups share a large amount of software
- Such clients automatically form a global cache
Cross file(system) sharing

Global cooperative cache regardless of origin servers

- Two groups of clients accessing Slax/Knoppix servers
- Client groups share a large amount of software
- Such clients automatically form a global cache
Cross file(system) sharing

Global cooperative cache regardless of origin servers

- Two groups of clients accessing Slax/Knoppix servers
- Client groups share a large amount of software
- BitTorrent supports only per-file groups
Content-based chunking

- Single overlay for all filesystems and content-based chunks
- Chunks – Variable-sized blocks based on content
  - Chunks are better – Exploit commonalities across files
  - LBFS-style chunks preserved across versions, concatenations

[Muthitacharoen et al. A Low-Bandwidth Network File System. SOSP ’01]
P2P systems – Themes

- Usability – Exploiting P2P for a range of systems
- Cross file(system) sharing – Use other groups
- Security issues in such wide-area P2P systems
- Topology management – Arrange nodes efficiently
- Scheduling dissemination of chunks efficiently
Traditionally, server authenticated read requests using uids
Traditionally, server authenticated read requests using uids

Challenge – Interaction between distrustful 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
- Accomplish this without a complex PKI
- Keep the number of rounds to a minimum
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
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
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
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
possibility 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
Security protocol

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

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

- $\text{Auth}_C$ – Authenticator to prove receiver has token
  - $\text{Auth}_C = \text{MAC} (T_B, \text{“Auth C”}, C, P, R_C, R_P)$

- $K$ – Key to encrypt chunk contents
  - $K = \text{MAC} (T_B, \text{“Encryption”}, C, P, R_C, R_P)$
Security protocol

- R_C, R_P – Random nonces to ensure freshness
- Auth_C – Authenticator to prove receiver has token
- K – Key to encrypt chunk contents
Security protocol

- $RC, RP$ – Random nonces to ensure freshness
- $Auth_C$ – Authenticator to prove receiver has token
  - $Auth_C = MAC \left( T_B, "\text{Auth C}'', C, P, RC, RP \right)$
- $K$ – Key to encrypt chunk contents
  - $K = MAC \left( T_B, "\text{Encryption}'', C, P, RC, RP \right)$
Client can check integrity of downloaded chunk

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

Source should not send chunks to unauthorized clients

- Malicious clients cannot send correct $\text{Auth}_C$

Eavesdropper shouldn’t get chunk contents

- All communication encrypted with $K$
P2P systems – Themes

- Usability – Exploiting P2P for a range of systems
- Cross file(system) sharing – Use other groups
- Security issues in such wide-area P2P systems
- Topology management – Arrange nodes efficiently
- Scheduling dissemination of chunks efficiently
Topology management

- Fetch metadata from the server
- Look up clients caching needed chunks in overlay
- Overlay is common to all file systems
- Overlay organized like a DHT – Get and Put
Fetch metadata from the server
Look up clients caching needed chunks in overlay
Overlay is common to all file systems
Overlay organized like a DHT – Get and Put
Fetch metadata from the server
Look up clients caching needed chunks in overlay
Overlay is common to all file systems
Overlay organized like a DHT – Get and Put
Fetch metadata from the server
Look up clients caching needed chunks in overlay
Overlay is common to all file systems
Overlay organized like a DHT – Get and Put
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
  - $I_B = \text{MAC}(T_B, \text{“Indexing Key”})$
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 TCP connection to download more chunks
- Exchange mutually needed chunks w/o indexing overhead
Overlay – Locality awareness

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

[Freedman et al. Democratizing content publication with Coral. NSDI ’04]
BitTorrent
- Neighbours “fixed”
- Makes cross file sharing difficult

Shark
- Chunk-based overlay
- Neighbours based on data
- Enables cross file sharing
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
Emulab – 100 Nodes on LAN

- Shark – 88s
- SFS – 775s (≈ 9x better), NFS – 350s (≈ 4x better)
- SFS less fair because of TCP backoffs
Shark ≈ 7 min – 95th Percentile
SFS ≈ 39 min – 95th Percentile (5x better)
NFS – Triggered kernel panics in server
Shark vs SFS – 23 copies vs 185 copies (8x better)
- Overlay not responsive enough
- Torn connections, timeouts force going to server
Data served by Clients

- Maximum contribution $\approx 3.5$ copies
- Median contribution $\approx 0.75$ copies
P2P systems – Themes

- Usability – Exploiting P2P for a range of systems
- Cross file(system) sharing – Use other groups
- Security issues in such wide-area P2P systems
- Topology management – Arrange nodes efficiently

- Scheduling dissemination of chunks efficiently
Fetching Chunks – Order Matters

- Scheduling problem – What order to fetch chunks of 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
Random – 133s
Sequential – 203s
Random Wins !!! – 35% better
Performance characteristics of workloads

- For Shark, what matters is *throughput*
  - Operation should finish as soon as possible

- Another interesting class of applications
  - Video streaming – RedCarpet

- Study the scheduling problem for *Video-on-Demand*
  - Press a button, wait a little while, and watch playback
Challenges for VoD

- **Near VoD**
  - Small *setup time* to play videos
- **High sustainable goodput**
  - Largest slope osculating block arrival curve
  - Highest video encoding rate system can support

**Goal:** Do block scheduling to achieve low setup time and high goodput
System design – Outline

- Components based on BitTorrent
  - Central server (tracker + seed)
  - Users interested in the data
- Central server
  - Maintains a list of nodes
- Each node finds neighbours
  - Contacts the server for this
  - Another option – Use gossip protocols
- Exchange data with neighbours
  - Connections are bi-directional
All nodes have unit bandwidth capacity
  ▶ 500 nodes in most of our experiments
Each node has 4-6 neighbours
Simulator is round-based
  ▶ Block exchanges done at each round
  ▶ System moves as a whole to the next round
File divided into 250 blocks
  ▶ Segment is a set of consecutive blocks
  ▶ Segment size = 10
Maximum possible goodput – 1 block/round
Why 95\textsuperscript{th} percentile?

- (x, y)
  - y nodes have a goodput at least x
- Red line at top shows 95\textsuperscript{th} percentile
- 95\textsuperscript{th} percentile a good indicator of goodput
  - Most nodes have at least that goodput
Feasibility of VoD – What does block/round mean?

- 0.6 blocks/round with 35 rounds of setup time
- Two independent parameters
  - 1 round \( \approx \) 10 seconds
  - 1 block/round \( \approx \) 512 kbps
- Consider 35 rounds a good setup time
  - 35 rounds \( \approx \) 6 min
- Goodput = 0.6 blocks/round
  - Max encoding rate = 0.6 \* 512 > 300 kbps
- Size of video = 250/0.6 \( \approx \) 70 min
Naïve scheduling policies

- **Segment-Random policy**
  - Divide file into segments
  - Fetch blocks in random order inside same segment
  - Download segments in order

<table>
<thead>
<tr>
<th>Random</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Segment-Random</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>
Naïve approaches – Random

- Each node fetches a block at random
- **Throughput** – High as nodes fetch disjoint blocks
  - More opportunities for block exchanges
- **Goodput** – Low as nodes do not get blocks in order
  - 0 blocks/round
Naïve approaches – Sequential

- Each node fetches blocks in order of playback
- **Throughput** – Low as fewer opportunities for exchange
  - Need to increase this

Graph:
- **Setup Time (in rounds)**
- **Goodput (blocks/round)**
- 0.2 blocks/round
- Sequential
- Random
Naïve approaches – Segment-Random policy

- **Segment-Random**
  - Fetch random block from the segment the earliest block falls in
  - Increases the number of blocks propagating in the system

Graph showing:
- Setup Time (in rounds) on the x-axis
- Goodput (blocks/round) on the y-axis

- Segment-Random: 0.4 blocks/round
NetCoding – How it helps?

- A has blocks 1 and 2
- B gets 1 or 2 with equal prob. from A
- C gets 1 in parallel
- If B downloaded 1, link B-C becomes useless

- Network coding routinely sends $1 \oplus 2$
Network coding – Mechanics

- Coding over blocks \(B_1, B_2, \ldots, B_n\)
- Choose coefficients \(c_1, c_2, \ldots, c_n\) from finite field
- Generate encoded block:
  \[ E_1 = \sum_{i=1}^{n} c_i \times B_i \]

- Without \(n\) coded blocks, decoding cannot be done
  - Setup time at least the time to fetch a segment

[Gkantsidis et al. Network Coding for Large Scale Content Distribution. InfoCom '05]
Benefits of network coding

- Segment-Netcoding
- Segment-Random
- Sequential
- Random

Goodput
- ≈ 0.6 blocks/round with netcoding
- 0.4 blocks/round with seg-rand
Segment scheduling

- Netcoding good for fetching blocks in a segment, but cannot be used across segments
  - Because decoding requires all encoded blocks

- **Problem**: How to schedule fetching of segments?
  - Until now, sequential segment scheduling
  - Results in rare segments when nodes arrive at different times

- **Segment scheduling algorithm to avoid rare segments**
  - Evaluation with C# prototype
Segment scheduling

- A has 75% of the file
- Flash crowd of 20 Bs join with empty caches
- Server shared between A and Bs
  - Throughput to A decreases, as server is busy serving Bs
- Initial segments served repeatedly
  - Throughput of Bs low because the same segments are fetched from server
Segment scheduling – Problem

- 8 segments in the video; A has 75% = 6 segments, Bs have none.
- **Green** is popularity 2, **Red** is popularity 1.
  - Popularity = # full copies in the system.
### Segment scheduling – Problem

<table>
<thead>
<tr>
<th>Segment #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>G</td>
<td>G</td>
<td>G</td>
<td>G</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Bs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Popularities</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- Instead of serving A...
## Segment scheduling – Problem

<table>
<thead>
<tr>
<th>Segment #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Popularities</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- The server gives segments to Bs
Segment scheduling – Problem

- A’s goodput plummets
- Get the best of both worlds – Improve A and Bs
  - Idea: Each node serves only rarest segments in system
### Segment scheduling – Algorithm

When dst node connects to the src node...

- Here: dst is B, src is server

<table>
<thead>
<tr>
<th>Segment #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Popularities</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Segment scheduling – Algorithm

<table>
<thead>
<tr>
<th>Segment #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Popularities</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- src node sorts segments in order of popularity
  - Segments 7, 8 least popular at 1
  - Segments 1-6 equally popular at 2
Segment scheduling – Algorithm

<table>
<thead>
<tr>
<th>Segment #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Popularities</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- src considers segments in sorted order one-by-one and serves dst
  - Either completely available at src, or
  - First segment required by dst
Segment scheduling – Algorithm

<table>
<thead>
<tr>
<th>Segment #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Popularities</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Server injects blocks from segment needed by A into system
  - Avoids wasting bandwidth in serving initial segments multiple times
How does src figure out popularities?

- Centrally available at the server
  - Our implementation uses this technique
- Each node maintains popularities for segments
  - Could use a gossiping protocol for aggregation
Segment scheduling

Note that goodput of both A and Bs improves
Conclusions

- **Problem**: Designing scalable data-intensive applications
- Enabled low-end servers to scale to large numbers of users
  - Shark: Scales file server for read-heavy workloads
  - RedCarpet: Scales video server for near-Video-on-Demand
- Exploited P2P for file systems, solved security issues
- Techniques to improve scalability
  - Cross file(system) sharing
  - Topology management – Data-based, locality awareness
  - Scheduling of chunks – Network coding