On Scheduling of Peer-to-Peer Video Services

CS587x Lecture
Department of Computer Science
Iowa State University

Server Design

- Each server maintains a waiting queue Q and all requests \( (C_i, V_i) \) are first appended in the queue.
- When a server has sufficient resource, it schedules a pending request for service.
- The requests in the queue are served according to their arriving order, i.e., FIFO.
- Given a request \( (C_i, V_i) \) in the queue of a server, we can estimate its remaining waiting time \( T_i = f(B, D, R) \):
  - \( B \): the server bandwidth, i.e., the number of concurrent video streams it can sustain.
  - \( D \): the remaining data that needs to be sent for the requests that are being served.
  - \( R \): the service requests that arrive ahead of \( (C_i, V_i) \) and are still waiting for services in the queue.
Service Scheduling

Problem
- Given a video request, which server should be used to serve it

Goal
- Minimize service latency
  - Defined to be the period from the time when a client submits its request to the time when the client starts to download the video

Naïve Solution

1. Find a set of server candidates
   - A client \( c \) requesting a video \( v \) can send command \( \text{Search}(c, v) \), which will return a set of hosts that cache the video
   - The underlying file lookup mechanism determines which servers can be found

2. Choose a server
   - From the set of server candidates, the client checks each of them to find their current workload
   - The client selects the server that can provide the service at the earliest time

3. Submit request to the server
   - The client sends command \( \text{Request}(c, v) \) to the server
Naïve Solution Falls Short

Problem 1: while a video may be cached by many hosts in the entire system, a client requesting for the video may find only a few, typically one, of them
- These hosts may not be able to serve the client at the earliest possible time
- Cause: the inherent problem of lookup services

Problem 2: given a series of client requests, their choosing of servers should depend on their arriving order
- Different matches of clients and servers may result in significantly different performance results
- Cause: the set of videos cached by a server partially overlaps with those by other servers
Naïve Solution Falls Short

Two free servers: S1{V1, V2}, S2{V2, V3}
Two client requests: (C1, V1), (C2, V2)

**Case I**
C1 chooses S1
Result: C1 can be served immediately, but C2 needs to wait until C1 is finished

**Case II**
C1 chooses S2
Result: Both C1 and C2 can be served immediately

A Possible Solution

- Basic idea: Dynamically adjust the service schedule as requests arrive
  - Before a client submits its request to a server S, the client contacts the server for all of its pending requests
  - For each pending request (Ci, Vi), the client tries to find another server that can serve Ci at an earlier time
  - If there is such a server S’, request (Ci, Vi) is removed from server S and submitted to server S’

- Why this works
  - For problem 1
    - Clients looking for the same video may find different sets of servers, since the video lookups launched by these clients are from different spots in the system
  - For problem 2
    - A mismatch can now be corrected as new requests arrive
Challenge: Chain of Rescheduling

- To reschedule a pending request \((C_i, V_i)\), a client needs to find a server that can serve the request at an earlier time.
- The servers found by the client may be busy as well and none of them can serve \((C_i, V_i)\) at an earlier time.
- The client may try to reschedule the pending requests on each server.
  - This may result in a chain of rescheduling.

1. Client \(C\) requesting \(V_2\) finds \(S_1\), but \(S_1\) needs to serve \((C_1, V_1)\) first.
2. \(C\) tries to reschedule \((C_1, V_1)\) and finds \(S_2\), but \(S_2\) needs to serve \((C_2, V_3)\) first.
3. \(C\) tries to reschedule \((C_2, V_3)\), and so on.

Client Design

1. Building Closure Set
2. Shaking Closure Set
3. Executing Shaking Plan
Building Closure Set

1. Set ClosureSet=NULL, ServerSet=NULL and VideoSet=NULL
2. Send command Search(v)
   - For each server found, add it to ServerSet
3. While ServerSet != NULL
   - Remove a server, say S, from ServerSet
   - Add S to ClosureSet
   - Contact S for its waiting queue Q
   - For each request (Ci, Vi) in Q, if Vi is not in VideoSet, then
     - Send command Search (Vi)
     - If a server found is not in ServerSet, add it to ServerSet
     - Add Vi to VideoSet
Shaking Closure Set

- Given a set of servers and their service queues, the client tries to find a *shaking plan* that can minimize its service latency
  - A shaking plan is an ordered list of action items, each denoted as $T([C, V], S, S')$, meaning that “transferring [C, V] from S to S’”.
- The client can try each server as the start point of shaking
  - For each request $(C_i, V_i)$, try to find another server that can serve it at an earlier time

An Example

Requests arrive in this order

$S_1 \{V_1, V_2, V_x\}$

$S_2 \{V_2\}$

$S_3 \{V_1, V_2\}$

$S_4 \{V_3, V_4\}$

$S_5 \{V_4\}$

Shaking Handle

$S_1 \{V_1, V_2, V_x\}$

$S_2 \{V_2\}$

Shaking Action $T([C_2, V_2], S_1, S_2)$

Shaking Pool
An Example

Requests arrive in this order:
- $S_1: \{V_p, V_2, V_4\}$
- $S_2: \{V_2\}$
- $S_3: \{V_4\}$

Shaking Pool:
- $S_1, S_2, S_4$

Shaking Set ($C_1, V_1$) = $\{S_1\}$
Shaking Set ($C_2, V_2$) = $\{S_3\}$
Shaking Set ($C_3, V_3$) = $\{S_2\}$

Shaking Action:
- $T(C_2, V_2, S_1, S_2)$
- $T(C_3, V_3, S_2, S_3)$
- $T(C_4, V_4, S_1, S_3)$

An Example

Requests arrive in this order:
- $S_1: \{V_1, V_2, V_4\}$
- $S_2: \{V_2\}$
- $S_3: \{V_4\}$

Shaking Pool:
- $S_1$

Shaking Set ($C_1, V_1$) = $\{S_3\}$
Shaking Set ($C_2, V_2$) = $\{S_2\}$
Shaking Set ($C_3, V_3$) = $\{S_4\}$

Shaking Action:
- $T(C_1, V_1, S_3)$
- $T(C_3, V_3, S_2, S_3)$
- $T(C_4, V_4, S_1, S_3)$

$S_5$
Shaking([C, V], S)

1: Get SPool(V)
2: S' <= \{s | s ∈ SPool(V) and latency([C, V], s) ≤ latency([C, V], S) and latency([C, V], s) is the least among SPool(V)\}
3: if S' ≠ 0 then
4:   Add([C, V], S, S') to SPlan
5:   return S
6: else
7:   ShakingSet([C, V]) = \{s | s ∈ SPool(V) and s ∉ SShakingPool\}
8: if ShakingSet([C, V]) = 0 then
9:   return NULL;
10: end if
11: ShakingPool = SShakingPool \cup ShakingSet([C, V])
12: for all \{s | s ∈ ShakingSet([C, V])\} do
13:   for all [C_s, V_s] ∈ G(s) do
14:     \[Shake([C_s, V_s], s, ShakingPool)\]
15:     if latency([C_s, V_s]) ≠ NULL then
16:       Append \{T([C_s, V_s], s, Destination([C_s, V_s]))\} to SPlan
17:     end if
18: end for
19: end for
20: end if
21: end for
22: SShakingPool = ShakingPool - \{s\}
23: end for
24: return NULL;
25: end if

Executing Shaking Plan

- For each action T([C, V], S, S'),
  - the client sends server S a message Transfer([C, V], S')
  - Upon receiving such a request, S first checks if [C, V] is still in its queue
    - If it is not, S sends an Abort message to C
    - Otherwise, S sends a message Add([C, V], L) to S', where L is the expected service latency of [C, V] at S
  - When S' receives Add([C, V], L), it checks if it can serve [C, V] in the next L time unit
    - If not, sends an Abort message to S, which forwards to the client
    - Otherwise, add [C, V] in its queue and sends an OK message to S'
      - S' will remove [C, V] from its queue and inform the client
Advantages

1. For the shaking client
   - By rescheduling early service request, the current client itself can be served at an earlier time
   - Each time a service request is rescheduled, the request will be served at an earlier time

2. For the entire system
   - Each shake may balance only the server workload within a small region, the combination of shaking effort, by each client, can balance the workload of the entire system
   - What factors determine the shaking scope?

Performance Study

- **Performance Metrics**
  - Average service latency

- **Factors to investigate**
  - Video request rate
  - Number of servers

- **Default parameters**
  - 100 videos, each 1 hours
  - The popularity of these videos follows a zipf distribution, skep = 0.5
**Shaking Improvement**

- The shaking discussed above is greedy
  - A request is not rescheduled unless the request can be served at an earlier time
- Does greedy shaking result in the best performance?

**Motivation Example**

If transferred from S1 to S2, C1 suffers if $T_1 < T_2$. However, the service requests behind (C1, V1) will all benefit, i.e., the service latency for each request reduced $|V1|$. 

---

Expected service time for C1 is $T_1$

(C1, V1)

S1

Expected service time for C1 is $T_2$

S2
Observation

- Expected service time for C1 is T1
- Expected service time for C1 is T2

The average service latency is not minimized if we try to serve each request at the earliest possible time!!

Grace Shaking

1. To transfer a request, calculate:
   - Cost: \((T2-T1)^a\)
   - Gain: \(N*|V1|\)
2. If Cost < Gain, let’s do it.