CSCI 1330
Review Session
MapReduce (failure/perf + Big Data)
MapReduce = Big data frame that manages servers + data orchestration

Uses heartbeats to detect failures

\( \text{failures} \leftarrow \text{performance} \)
\( \Rightarrow \) progress reports to detect fast/slow servers

1. Why are servers slow?
2. How to recover/improve performance?
Which of these signals is the best method for detecting such a slow task?

(X) Progress reports
( ) Heartbeats
( ) CPU Utilization
( ) None of the above

If the slow task is due to data-imbalance, which of these approaches will best address the problem?

( ) Cancel and retry the task on a different server
( ) Duplicate (or clone) the task to a different server
( ) Quarantine the server (which is currently running the task), and never use it again.
(X) None of the above.
Network (TCP/UDP/RPC)

TCP V. UDP

<table>
<thead>
<tr>
<th>Connection oriented</th>
<th>No startup latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good features</td>
<td>Re-transmit/Retry</td>
</tr>
<tr>
<td></td>
<td>Re-order packets</td>
</tr>
</tbody>
</table>

You get Nothing

- Very fast
- Has no overhead

When to use?

1. Large bulk transfer
2. Web traffic

1. Small msg (RPC)
2. Live streaming

RPC

1. Semantic: At-most-once
   - 0 or 1 time
   - Requires complex tech

2. At-least-once
   - 1 or more times
   - Idempotent

Exactly once

- The func call happens once
- This is not practical

Retrying

- Duplicate suppression
- Response replay

Success
For short RPC messages

( ) TCP
(X) UDP

For live streaming events

( ) TCP
(X) UDP

Which of these is the main reason why UDP is used for RPC messages

( ) UDP provides amazing encoding that reduces data size
(X) TCP has a latency overhead but UDP does not
( ) TCP attempts to provide fairness which is bad in data centers
( ) UDP uses special sockets that are optimized for distributed systems
Which of these filesystem calls do you expect to be idempotent? (Select all that apply)

Note: these call do not exactly correspond to C calls and I have provided a description of their semantics.

[X] Set(key, value) // this function store the “value” in memory under the identifier “key”

[ ] fgets(char *str, int n, FILE *stream) // this function reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

[X] Delete(key) // This function deletes the data that was stored under the “key”.

You want to design a RPC framework that provides at-least-once semantics. Your team informs you that you only need to implement the 'request-retry' feature and you do not need to implement either 'duplicate suppression' or 'response replay'. Which of these best explain why you can ignore those two?

( ) 'request-retry' by default provides 'duplicate suppression' and 'response replay'.
(X) Neither 'duplicate suppression' or 'response replay' are needed because at-least-once call can support multiple invocations.
( ) The team is wrong and you need to implement all three features.
Load Balancing (Global / Local)

1. DNS: name to IP mapping
2. Network to get to cluster (BGP)
3. LB maps to server
   (Local LB) → distribute load between servers

Global LB
1. Reduce latency (get you a closer cluster)
2. Data policy
Global LB

Goals

1. Reduce latency to client/cluster
2. Respect data policy

Techniques

0. DNS
2. BGP

Local LB

Goals

1. Distribute load between servers

Techniques

0. Round Robin
2. Random
3. modulo hash
4. Consistent hashing

Local LB

Key issue with Round Robin/Random is that they send connection to different servers. This is bad because we want all connection from a user to go to same server (TCP overhead/session info)

Consistent hashing → consistently assigns client/user to the same server
\[ D = \text{hash (string)} \mod N \]

\[ \text{size of the key space} \]

0

1

4/3

5/3

S

S

MDS

131

\[ \text{IP address:port} \]

4/3

S

\[ k \]

\[ \text{MD5} \]

\[ 143 \]

\[ 916 \]
In this system if one of the $N$ server crashes, how many clients will need to be moved?

( ) $K$ (all clients)
( ) $K/N$ (only the clients on a server) [X]
( ) 0 (no client will need to be moved)
( ) $K/(2N)$ (half of the clients on a server)
( ) $N$ (one client from each of the servers)

In this system, if a new server is added to the system crashes (now there are $N+1$ server), how many clients will need to be moved?

( ) $K$ (all clients)
( ) $K/N$ (the num of clients on a server)
( ) 0 (no client will need to be moved) [X]
( ) $K/(2N)$ (half of the clients on a server)
( ) $N$ (one client from each of the servers)

Which of these enables the global DNS infrastructure to scale?

[ ] Client side caching of responses
[X] The use of TCP
[X] Hierarchical partitioning of the Name space
[ ] Automated partitioning of the IP space (Divide IP addresses and allocate different IP addresses to different cities)

Which of these are potential goals for a load balancer within a cluster?

[X] evenly distribute work
[X] ensure consistent map of client to server
[ ] assign client to server based on location
[ ] provide local RPC semantics
The max # of hops in any lookup is \( O(\log N) \).
<table>
<thead>
<tr>
<th>Grateful</th>
<th>Un Graceful</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Update routing tables of nodes in your backpointer structure</td>
<td>1. Heartbeats</td>
</tr>
<tr>
<td>2. Move objects to new root locations</td>
<td>2. Bad nodes detected via RPC calls</td>
</tr>
</tbody>
</table>
For Graceful failures, what is the main technique that tapestry provides to ensure this guarantee:

( ) Clients republish object
(X) Nodes (i.e., server) transfer object store before failure
( ) Nodes (i.e., servers) use backpointers to find objects
( ) Nodes (i.e., servers) use route table to find clients

For NON-Graceful failures, what is the main technique that tapestry provides to ensure this guarantee:

(X) Clients republish object
( ) Nodes (i.e., server) transfer object store before failure
( ) Nodes (i.e., servers) use backpointers to find objects
( ) Nodes (i.e., servers) use route table to find clients
Chord

client ID $\rightarrow$ next largest server ID
Time + Global State

- Real Time is Bad (it is not monotonic/hard to synchronize servers)
- What you care about is ordering of events
  - Total
    - every server agrees on the order
  - FIFO
    - servers don't agree on all events: events need to order events based on clock/ID from individual servers
  - Causal
    - Vector Clocks
      - "happens after" relationship

Logical clocks

Global State (distributed snapshots)

- Time based; manual; continuous; Chandy-Lamport

- Time synchronization: time to go to all servers
- Only valid snapshot maybe at the beginning of the system
- Very strong assumptions:
  1. No server will fail
  2. The n/w never reorders msgs
Consistent snapshots: \( \text{iff} \) for every "read" event in the distributed snapshot you also have the "send" event.

Use of Vector Clocks to detect inconsistencies:

\[ VC_{A,B} \]

\[ VC_{A,B} > VC_{B,B} \]

For the latest events in the snapshot, check against some server. If server A has a higher clock for server B than server B has for itself.

\[ [10,10] \text{ Snapshot} \]

\[ [5,6] \text{ Snapshot} \]

\[ [5,10] \text{ Snapshot} \]
Consider the following system with two servers S1 and S2.

S1 receives and process events in the following order: X1, X2.
S2 receives and process events in the following order: Z1, Z2.

Which of these ordering of events is total but not FIFO ordered:

( ) S1: X1, Z1, X2, Z2   S2: X1, Z1, X2, Z2
( ) S1: X1, X2, Z1, Z2   S2: X1, Z1, X2, Z2
( ) S1: X1, X2, Z1, Z2   S2: Z1, Z2, X1, X2
(X) S1: X1, Z2, X2, Z1   S2: X1, Z2, X2, Z1

Which of these ordering of events is both total and FIFO ordered:

(X) S1: X1, Z1, X2, Z2   S2: X1, Z1, X2, Z2
( ) S1: X1, X2, Z1, Z2   S2: X1, Z1, X2, Z2
( ) S1: X1, X2, Z1, Z2   S2: Z1, Z2, X1, X2
( ) S1: X1, Z2, X2, Z1   S2: X1, Z2, X2, Z1
consistent

This is not consistent

snapshot @ P2 doesn't include the send event associated with "M"
Topics

- MapReduce (Liteminer)
- Networking (TCP/UDP, P2P)
- L/B (Local vs. Global/Consistent hash)
- Time (logical/Vector/Snapshots)