CSCI 1370
Review Session
Map Reduce (failure/perf + Big Data)
MapReduce = Big data framework that manages servers and orchestrates data.

Uses heartbeats to detect failures

\[ \text{Performance} \rightarrow \text{progress reports to detect fast/slow servers} \]

\[ \text{failures} \leftarrow \text{MapReduce} \]

(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 vs. UDP

<table>
<thead>
<tr>
<th>Connection oriented</th>
<th>You get Nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good features</td>
<td>Very fast</td>
</tr>
<tr>
<td>Retransmit Retry</td>
<td>has no overhead</td>
</tr>
<tr>
<td>Reorder packets</td>
<td></td>
</tr>
</tbody>
</table>

when to use?

1. Large bulk transfer
2. Web traffic
3. Small msg (RPC)
4. Live streaming

RPC

1. Semantic: at-most-once → at-least-once → exactly once
2. 0 or 1 time → 1 or more times → Idempotent
   - Requires complex tech
   - The func call happens once
   - This is not practical

- Retry
- Duplicate suppression
- Response replay
For short RPC messages

( ) TCP
( ) UDP

For live streaming events

( ) TCP
( ) UDP

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

( ) UDP provides amazing encoding that reduces data size
( ) 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 stores the “value” in memory under the identifier “key”
[ ] char *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
   - Brown routes

2. Network
   - Routes to brown servers

3. LB maps to server
   - LB maps to cluster
   - Cluster maps to server

- Global LB
  - Reduce latency (get you a close cluster)
  - Data policy

- Local LB
  - Distribute load to servers
### Global LB vs Local LB

<table>
<thead>
<tr>
<th>Goals</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. reduce latency to client/cluster</td>
<td>1. distribute load btw servers</td>
</tr>
<tr>
<td></td>
<td>2. respect data policy</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Techniques</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. DNS</td>
<td>FB</td>
</tr>
<tr>
<td></td>
<td>2. BGP</td>
<td>Azure</td>
</tr>
</tbody>
</table>

**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
\[ D = \text{hash(string)} \mod N \]
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)
( ) 0 (no client will need to be moved)
( ) K/(2*N) (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)
( ) K/(2*N) (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?

[X] Client side caching of responses
[ ] 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
Each row guaranteed to have at least one entry; the note itself.

The max # of hops in any lookup is $O(\log_2 N)$.

# of DIGIT in ID; every hop takes you one digit closer to destination.

Backpointers:

- Grateful exit: use backpointers to determine which notes to update.

Tapestry

$\log_2 N$}

$\beta$

size of routing table

$C \cdot \log_2 N \cdot \beta$

# of slots
<table>
<thead>
<tr>
<th>Graceful</th>
<th>Un Graceful</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. update routing tables of nodes in your backpointer structure</td>
<td>1. Send 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
( ) 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:

( ) 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 → 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

↓

FIFO

Total

every server agrees on the order

Causal

servers don't agree on all events: events need to order events based on clock/ID from individual servers

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 0 the beginning of the system

very strong assumptions

1) no server will fail

2) the n/w never reorders msgs
Consistent snapshots: \textit{iff} for every "reco" event in the distributed snapshot you also have the "send" event.

Use of Vector Clocks to detect inconsistencies:

\[ V_C[A] > V_C[B] \]

If server A has higher clock for server B then server A has for itself.
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, X1, Z2

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

Map Reduce (Listeminer)

Networking (TCP/UDP, RCE)

L/B (Local V. Global/consistent hash)

Time (logical/Vector/snapshots)