Announcements

• SDSC data center visit: Mar 22, 11am
• Project 2:
  – New server assignments via email
  – Copy off all files from AWS machines by Wed@5p
    • Otherwise files will be lost
• Today:
  – RPC → REST
  – Replicating data: how to ensure consistency?
REST
(REpresentational State Transfer)
Background

• Roy Fielding Ph.D. dissertation, 2000
  – “Architectural Styles and the Design of Network-Based Software Architectures”

• RPC-like interface to web services
  – Based on HTTP, URLs, JSON, XML

• Access and update remote state via HTTP

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Request to read a Web page</td>
</tr>
<tr>
<td>HEAD</td>
<td>Request to read a Web page’s header</td>
</tr>
<tr>
<td>PUT</td>
<td>Request to store a Web page</td>
</tr>
<tr>
<td>POST</td>
<td>Append to a named resource (e.g., a Web page)</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove the Web page</td>
</tr>
<tr>
<td>TRACE</td>
<td>Echo the incoming request</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Query certain options</td>
</tr>
</tbody>
</table>
HTTP + RPC?

+ ?? = RPC?

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
</tr>
<tr>
<td>HEAD</td>
</tr>
<tr>
<td>PUT</td>
</tr>
<tr>
<td>POST</td>
</tr>
<tr>
<td>DELETE</td>
</tr>
<tr>
<td>TRACE</td>
</tr>
<tr>
<td>CONNECT</td>
</tr>
<tr>
<td>OPTIONS</td>
</tr>
</tbody>
</table>

- Must represent application state in a way that can be exposed to HTTP
- Idea:
  - Encode state representation as a URL
Representing state as URLs

GET /Addressbook/contacts

[{
  "id":1,
  "name":"George Porter",
  "phone":"858-561-1234",
  "room number":3104
},
{
  "id":2,
  "name":"Stefan Savage",
  "phone":"858-561-8172",
  "room number":3105
}
...

Representing state as URLs

GET /Addressbook/contact/2

{  
  "id":2,
  "name":"Stefan Savage",
  "phone":"858-561-8172",
  "room number":3105
}

Representing state as URLs

POST /Addressbook/contacts

{  
  "id":3,  
  "name":"A. Turing",  
  "phone":"858-561-8212",  
  "room number":3108
}

Result:

200 OK
REST example
REST examples “In the wild”

**URI**

Use the following URI to obtain Contact Objects or add a Contact Object for a user identified by the `{guid}` value.

https://social.yahooapis.com/v1/user/{guid}/contacts

**HTTP OPERATIONS SUPPORTED**

- GET
- POST
- PUT

**QUERY PARAMETERS SUPPORTED**

- format
- view
- start
- count
Example: Twitter

- Every aspect of Twitter is available via REST interfaces
- Data encoded in JSON
  - Tweets, friends, Timelines, ...
- Developer interface to REST API via:
  - https://dev.twitter.com/rest/tools/console
Replicating data:
Consensus and updates
High Performance and Availability Through Replication?

- Improve probability that nearby replica can handle request
- Increase system complexity
The Need for Replication

• Certain mission critical Internet services must provide 100% availability and predictable (high) performance to clients located all over the world
  – With scale of the Internet, high probability that some replica/some network link unavailable at all times

• Replication is the only way to provide such guarantees
  – Despite any increased complexities, must investigate techniques for addressing replication challenges
Replication Goals

• Replicate network service for:
  – Better performance
  – Enhanced availability
  – Fault tolerance

• How could replication lower performance, availability, and fault tolerance?
Replication Challenges

• Transparency
  – Mask from client the fact that there are multiple physical copies of a logical service or object
  – Expanded role of naming in networks/dist systems

• Consistency
  – Data updates must eventually be propagated to multiple replicas
  – Guarantees about latest version of data?
  – Guarantees about ordering of updates among replicas?

• Increased complexity…
Replication Model

Client -> FE -> Replica -> Replica

Client -> FE -> Service

Replica -> Replica
How to Handle Updates?

• Problem: all updates must be distributed to all replicas
  – Different consistency guarantees for different services
  – Synchronous vs. asynchronous update distribution
  – Read/write ratio of workload

• Primary copy
  – All updates go to a single server (master)
  – Master distributes updates to all other replicas (slaves)

• Gossip architecture
  – Updates can go to any replica
  – Each replica responsible for eventually delivering local updates to all other replicas
Each replica must handle write load of entire system?
Update Ordering Requirements

• Total Order
  – Bulletin board: all messages assigned globally unique message identifier
  – For messages $r_1, r_2$, either $r_1$ appears before $r_2$ at all replicas or $r_1$ appears after $r_2$ at all replicas

• Causal Order
  – Bulletin board: message replies appear after original posting
  – For messages $r_1, r_2$, $r_1$ appears before $r_2$ if $r_1$ happens before $r_2$
Happens Before

- Captures potential causal ordering (information flow)
- a → b if a takes place before b in the same process
- Send(m) → recv(m)
- Transitivity holds

- Need tie-breaker such as IP address or MAC address
Implementing Total Ordering

• Use *sequencer*
  – Send updates to centralized site, assign monotonically increasing identifier, distribute to all replicas
  – Single point of failure, contention

• Distributed total ordering
  – Front end sends update to all replicas
  – Each replica proposes unique id
  – Front end picks highest value
  – Transmits final value back to replicas
  – 3 messages/replica overhead
Network Partitions

• Some failure (either network or host) keeps replicas from communicating with one another
• How to proceed with read/write transactions in case where not all replicas can be contacted?
Network Partitions

• Some failure (either network or host) keeps replicas from communicating with one another
• How to proceed with read/write transactions in case where not all replicas can be contacted?
  – Optimistic versus pessimistic techniques
• Optimistic: proceed as normal, resolving conflicts later
• Pessimistic: ensure that replicated database can eventually be restored to consistent state without user intervention
  – Assumes that partition will eventually be repaired
  – What if the network is never fully connected?
Quorum (Voting) Consensus Techniques

• Allow updates to be completed even when only a subset of replicas are available
  – Or have the latest version of a data item
  – May need to bring some set of replicas up to date to proceed

• One technique is the weighted voting scheme
  – Assign a number of votes to each replica
  – Determine a write quorum $W$ and a read quorum $R$
    • $W > \text{half the total votes}$, $R+W > \text{total number of votes}$
  – Ensures that there is some overlap between read and write quorums
    • Read operation guaranteed access to one site with latest version
Quorum Example

Write quorum

Read quorum
Quorum Consensus

• Write operations can be propagated in background to replicas not in quorum
  – Assumes eventual repair of any network partition
• Operations are slowed by the necessity of first gathering a quorum
  – Though previously, all writes had to go to all replicas
    • With quorum system, must only contact subset of replicas
Quorum Example

- 5 replicas, read quorum: 5, write quorum: 1
  - $R+W > 5$ votes ensures overlap between any read/write quorum
- How does this perform for reads?
- How does this perform for writes?
Quorum Example

- 5 replicas, read quorum: 1, write quorum: 5
  - R+W>5 votes ensures overlap between any read/write quorum
- How does this perform for reads?
- How does this perform for writes?
Quorum Example

- 5 replicas, read quorum: 3, write quorum: 3
  - R+W>5 votes ensures overlap between any read/write quorum
- How does this perform for reads?
- How does this perform for writes?
Managing concurrency

• What happens if $W=1$, $R=5$, and two clients concurrently update a variable?
  – Might choose different and orthogonal write quorums

• Solution 1:
  – Synchronization outside of the system

• Solution 2:
  – Choose $W > (N/2)$ where $N$=# nodes