Remote Procedure Call (RPC)

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive

- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    - `remoteFileSystem->Read("rutabaga")`
  - Translated automatically into call on server:
    - `fileSys->Read("rutabaga")`

RPC Implementation

- Request-response message passing (under covers!)
- “Stub” provides glue on client/server
  - Client stub is responsible for “marshalling” arguments and “unmarshalling” the return values
  - Server-side stub is responsible for “unmarshalling” arguments and “marshalling” the return values.

- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow
RPC Details (1/3)

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)

- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an “interface definition language (IDL)”
    » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    » Code for server to unpack message, call procedure, pack results, send them off

RPC Details (2/3)

- Cross-platform issues:
  - What if client/server machines are different architectures/ languages?
    » Convert everything to/from some canonical form
    » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint
    (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    » This is another word for “naming” at network level
    » Static: fixed at compile time
    » Dynamic: performed at runtime

RPC Details (3/3)

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    » Name service provides dynamic translation of service ⇔ mbox
  - Why dynamic binding?
    » Access control: check who is permitted to access service
    » Fail-over: If server fails, use a different one

- What if there are multiple servers?
  - Could give flexibility at binding time
    » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    » Choose unloaded server for each new request
    » Only works if no state carried from one call to next

- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC: Non-Atomic Failures

- Different failure modes in dist. system than on a single machine
- Consider many different types of failures
  - User-level bug causes address space to crash
  - Machine failure, kernel bug causes all processes on same machine to fail
    » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    » Did my cached data get written back or not?
    » Did server do what I requested or not?
- Answer? Distributed transactions/Byzantine Commit
**Problems with RPC: Performance**

- Cost of Procedure call « same-machine RPC « network RPC
- Means programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex

**Administrivia**

- Midterm 3 coming up on **Mon 4/24 6:30-8PM**
  - All topics up to and including Lecture 15
    - Focus will be on Lectures 16 – 23 and associated readings, and Projects 3
    - But expect 20-30% questions from materials from Lectures 1-15
  - A-L 245 Li Ka Shing, M-S 2060 VLSB, T-Z 2040 VLSB
  - Closed book
  - A single hand-written note, both sides

**Key Value Storage**

- Handle huge volumes of data, e.g., PetaBytes!
  - Store (key, value) tuples

- Simple interface
  - `put(key, value);` // insert/write “value” associated with “key”
  - `value = get(key);` // get/read data associated with “key”

- Used sometimes as a simpler but more scalable “database”
Key Values: Examples

- **Amazon:**
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, …)

- **Facebook, Twitter:**
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)

- **iCloud/iTunes:**
  - Key: Movie/song name
  - Value: Movie, Song

Key-Value Storage Systems in Real Life

- **Amazon**
  - DynamoDB: internal key value store used for Amazon.com (cart)
  - Simple Storage System (S3)

- **BigTable/HBase/Hypertable:** distributed, scalable data store

- **Cassandra:** “distributed data management system” (developed by Facebook)

- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)

- **BitTorrent:** distributed file location: peer-to-peer sharing system

Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-values across many machines

Challenges

- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s
Key Questions

• put(key, value): where to store a new (key, value) tuple?

• get(key): where is the value associated with a given “key” stored?

• And, do the above while providing
  – Fault Tolerance
  – Scalability
  – Consistency

Directory-Based Architecture

• Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys

Master/Directory

put(K14, V14)

get(K14)

V14

N1 N2 N3 N50

K105 V105

K5 V5 K14 V14

...
**Directory-Based Architecture**

- Having the master relay the requests → **recursive query**
- Another method: **iterative query**
  - Return node to requester and let requester contact node

**Discussion: Iterative vs. Recursive Query**

- **Recursive Query:**
  - Advantages:
    - Faster, as typically master/directory closer to nodes
    - Easier to maintain consistency, as master/directory can serialize puts() and gets()
  - Disadvantages: scalability bottleneck, as all “Values” go through master/directory
- **Iterative Query**
  - Advantages: more scalable
  - Disadvantages: slower, harder to enforce data consistency

**Fault Tolerance**

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures
Fault Tolerance

• Or we can use recursive query and iterative replication...

<table>
<thead>
<tr>
<th>K14</th>
<th>V14</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td></td>
</tr>
<tr>
<td>N2</td>
<td></td>
</tr>
<tr>
<td>N3</td>
<td></td>
</tr>
<tr>
<td>N50</td>
<td></td>
</tr>
</tbody>
</table>

scalability

• More Requests:
  – Can serve requests from all nodes on which a value is stored in parallel
  – Master can replicate a popular value on more nodes

• Master/directory scalability:
  – Replicate it
  – Partition it, so different keys are served by different masters/directories
    » How do you partition?

Scalability: Load Balancing

• Directory keeps track of the storage availability at each node
  – Preferentially insert new values on nodes with more storage available

• What happens when a new node is added?
  – Cannot insert only new values on new node. Why?
  – Move values from the heavy loaded nodes to the new node

• What happens when a node fails?
  – Need to replicate values from fail node to other nodes

Consistency

• Need to make sure that a value is replicated correctly

• How do you know a value has been replicated on every node?
  – Wait for acknowledgements from every node

• What happens if a node fails during replication?
  – Pick another node and try again

• What happens if a node is slow?
  – Slow down the entire put()? Pick another node?

• In general, with multiple replicas
  – Slow puts and fast gets
Consistency (cont’d)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order
  - put(K14,V14') and put(K14,V14'') reach N1 & N3 in reverse order
  - What does get(K14) return?
    - Undefined!

Large Variety of Consistency Models

- Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  - Think “one updated at a time”
  - Transactions

- Eventual consistency: given enough time all updates will propagate through the system
  - One of the weakest form of consistency; used by many systems in practice
  - Must eventually converge on single value/key (coherence)

- And many others: causal consistency, sequential consistency, strong consistency, …

Quorum Consensus

- Improve put() and get() operation performance

- Define a replica set of size N
  - put() waits for acknowledgements from at least W replicas
  - get() waits for responses from at least R replicas
  - W+R > N

- Why does it work?
  - There is at least one node that contains the update

- Why might you use W+R > N+1?
Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer

![Diagram of Quorum Consensus Example]

Scaling Up Directory

- Challenge:
  - Directory contains a number of entries equal to number of (key, value) tuples in the system
  - Can be tens or hundreds of billions of entries in the system!

- Solution: consistent hashing
  - Associate to each node a unique id in an uni-dimensional space $0..2^m-1$
  - Partition this space across $m$ machines
  - Assume keys are in same uni-dimensional space
  - Each (Key, Value) is stored at the node with the smallest ID larger than Key

Key to Node Mapping Example

- $m = 8 \rightarrow$ ID space: 0.63
- Node 8 maps keys [5,8]
- Node 15 maps keys [9,15]
- Node 20 maps keys [16, 20]
- ...
- Node 4 maps keys [59, 4]

![Diagram of Key to Node Mapping Example]

Scaling Up Directory

- With consistent hashing, directory contains only a number of entries equal to number of nodes
  - Much smaller than number of tuples

- Next challenge: every query still needs to contact the directory

- Solution: distributed directory (a.k.a. lookup) service:
  - Given a key, find the node storing value associated to the key

- Key idea: route request from node to node until reaching the node storing the request’s key

- Key advantage: totally distributed
  - No point of failure; no hot spot
Chord: Distributed Lookup (Directory) Service

- Key design decision
  - Decouple correctness from efficiency
- Properties
  - Each node needs to know about $O(\log(M))$, where $M$ is the total number of nodes
  - Guarantees that a tuple is found in $O(\log(M))$ steps
- Many other lookup services: CAN, Tapestry, Pastry, Kademlia, …

Lookup

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
- E.g., node=4 lookups for node responsible for Key=37

Stabilization Procedure

- Periodic operation performed by each node $n$ to maintain its successor when new nodes join the system

```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
    succ = x; // if x better successor, update
    succ.notify(n); // n tells successor about itself
  n.notify(n'); // n tells predecessor about itself
  if (pred = nil or n' ∈ (pred, n))
    pred = n'; // if n' is better predecessor, update
```

Joining Operation

- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15

```
succ=4
pred=44
```

```
succ=58
pred=35
```

```
succ=nil
pred=nil
```
§ \( n = 50 \) sends `join(50)` to node 15

§ \( n = 44 \) returns node 58

§ \( n = 50 \) updates its successor to 58

Joining Operation

- \( n = 50 \) executes `stabilize()`
  - \( x = 44 \)
  - \( \text{succ} = 58 \)

\[ \text{n.stabilize()} \]
\[ x = \text{succ.pred}; \]
\[ \text{if (} x \subseteq (n, \text{succ}) \text{)} \]
\[ \text{succ} = x; \]
\[ \text{succ.notify}(n); \]

Joining Operation

- \( n = 50 \) sends to its successor (58) `notify(50)`
  - \( \text{succ} = 58 \)
  - \( \text{pred} = 35 \)

\[ \text{n.stabilize()} \]
\[ x = \text{succ.pred}; \]
\[ \text{if (} x \subseteq (n, \text{succ}) \text{)} \]
\[ \text{succ} = x; \]
\[ \text{succ.notify}(n); \]
Joining Operation

- n=58 processes
- notify(50)
  - pred = 44
  - n' = 50

n.notify(n')
if (pred = nil or n'⊆(pred, n))
  pred = n'

Joining Operation

- n=58 processes
- notify(50)
  - pred = 44
  - n' = 50
  - set pred = 50

n.notify(n')
if (pred = nil or n'⊆(pred, n))
pred = n'

Joining Operation

- n=44 runs
- stabilize()
- n's successor (58) returns x = 50

n.stabilize()
x = succ.pred;
if (x⊆(n, succ))
succ = x;
succ.notify(n);

Joining Operation

- n=44 runs
- stabilize()
  - x = 50
  - succ = 58
Joining Operation

- n=44 runs stabilize()
- x = 50
- succ = 58
- n=44 sets succ=50

\[
\begin{align*}
\text{n.stabilize()} & : x = \text{succ.pred}; \\
& \quad \text{if (x} \subseteq (n, \text{succ}})) \\
& \quad \text{succ} = x; \\
& \quad \text{succ.notify(n);} \\
\end{align*}
\]

- n=44 sends notify(44) to its successor

\[
\begin{align*}
\text{n.notify(n')} & : \text{if (pred = nil or n} \subseteq (\text{pred}, n)) \\
& \quad \text{pred} = n' \\
\end{align*}
\]

Joining Operation

- n=50 processes notify(44)
- pred = nil
- n=50 sets pred=44
Joining Operation (cont’d)

- This completes the joining operation!

Achieving Efficiency: finger tables

Finger Table at 80

Say \( m = 7 \)

\[
\text{\( f(i) = (80 + 2^i) \mod 2^m \)}
\]

\( i \)th entry at peer with id \( n \) is first peer with id \( n + 2^i (\mod 2^m) \)

Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the \( k (> 1) \) immediate successors instead of only one successor

- In the \( \text{pred()} \) reply message, node A can send its \( k-1 \) successors to its predecessor B

- Upon receiving \( \text{pred()} \) message, B can update its successor list by concatenating the successor list received from A with its own list

- If \( k = \log(M) \), lookup operation works with high probability even if half of nodes fail, where \( M \) is number of nodes in the system

Storage Fault Tolerance

- Replicate tuples on successor nodes

- Example: replicate \((K14, V14)\) on nodes 20 and 32
Storage Fault Tolerance

- If node 15 fails, no reconfiguration needed
  - Still have two replicas
  - All lookups will be correctly routed
- Will need to add a new replica on node 35

Summary (1/2)

- Remote Procedure Call (RPC): Call proc on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments (in stub)
- Key-Value Store:
  - Two operations
    » put(key, value)
    » value = get(key)
  - Challenges
    » Fault Tolerance → replication
    » Scalability → serve get()'s in parallel; replicate/cache hot tuples
    » Consistency → quorum consensus to improve put() performance

Summary (2/2)

- Chord:
  - Highly scalable distributed lookup protocol
  - Each node needs to know about $O(\log(M))$, where $M$ is the total number of nodes
  - Guarantees that a tuple is found in $O(\log(M))$ steps
  - Highly resilient: works with high probability even if half of nodes fail