REMOTE PROCEDURE CALLS

George Porter
Jan 25, 2019

ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
WHY RPC?

- The typical programmer is trained to write single-threaded code that runs in **one place**
- **Goal:** Easy-to-program network communication that makes client-server communication **transparent**
  - Retains the “feel” of writing centralized code
  - Programmer needn’t think about the network
REMOTE PROCEDURE CALL (RPC)

• Distributed programming is challenging
  • Need common primitives/abstraction to hide complexity
  • E.g., file system abstraction to hide block layout, process abstraction for scheduling/fault isolation

• In early 1980’s, researchers at PARC noticed most distributed programming took form of remote procedure call

WHAT’S THE GOAL OF RPC?

• Within a single program, running in a single process, recall the well-known notion of a procedure call:
  • Caller pushes arguments onto stack,
    • jumps to address of callee function
  • Callee reads arguments from stack,
    • executes, puts return value in register,
    • returns to next instruction in caller

RPC’s Goal: To make communication appear like a local procedure call: transparency for procedure calls
RPC EXAMPLE

Local computing

\[ X = 3 \times 10; \]
\[ \text{print}(X) \]
\[ > 30 \]

Remote computing

\[ \text{server} = \text{connectToServer}(S); \]
\[ \text{Try:} \]
\[ \quad X = \text{server.mult}(3,10); \]
\[ \quad \text{print}(X) \]
\[ \text{Except e:} \]
\[ \quad \text{print} \text{ "Error!"} \]
\[ > 30 \]
\[ \text{or} \]
\[ > \text{Error} \]

RPC ISSUES

- Heterogeneity
  - Client needs to **rendezvous** with the server
  - Server must **dispatch** to the required function
    - What if server is **different** type of machine?
- Failure
  - What if messages get **dropped**?
  - What if client, server, or network **fails**?
- Performance
  - Procedure call takes \(\approx 10\) cycles \(\approx 3\) ns
  - RPC in a data center takes \(\approx 10\ \mu s\) (\(10^3\times\) slower)
    - In the wide area, typically \(10^6\times\) slower
PROBLEM: DIFFERENCES IN DATA REPRESENTATION

- Not an issue for local procedure call

- For a remote procedure call, a remote machine may:
  - Represent data types using different sizes
  - Use a different byte ordering (endianness)
  - Represent floating point numbers differently
  - Have different data alignment requirements
    - e.g., 4-byte type begins only on 4-byte memory boundary

BYTE ORDER

- x86-64 is a little endian architecture
  - Least significant byte of multi-byte entity at lowest memory address
    - “Little end goes first”
  - Some other systems use big endian
    - Most significant byte of multi-byte entity at lowest memory address
      - “Big end goes first”

```
int 5 at address 0x1000:
0x1000: 0000 0101
0x1001: 0000 0000
0x1002: 0000 0000
0x1003: 0000 0000
```

```
int 5 at address 0x1000:
0x1000: 0000 0000
0x1001: 0000 0000
0x1002: 0000 0000
0x1003: 0000 0101
```
PROBLEM: DIFFERENCES IN PROGRAMMING SUPPORT

• Language support varies:
  • Many programming languages have no inbuilt concept of remote procedure calls
    • e.g., C, C++, earlier Java
  • Some languages have support that enables RPC
    • e.g., Python, Haskell, Go

SOLUTION: INTERFACE DESCRIPTION LANGUAGE

• Mechanism to pass procedure parameters and return values in a machine-independent way
• Programmer may write an interface description in the IDL
  • Defines API for procedure calls: names, parameter/return types
• Then runs an IDL compiler which generates:
  • Code to marshal (convert) native data types into machine-independent byte streams
    • And vice-versa, called unmarshaling
  • Client stub: Forwards local procedure call as a request to server
  • Server stub: Dispatches RPC to its implementation
A DAY IN THE LIFE OF AN RPC

1. Client calls stub function (pushes params onto stack)

Client machine

Client process
  \( k = add(3, 5) \)

Client stub (RPC library)

A DAY IN THE LIFE OF AN RPC

1. Client calls stub function (pushes params onto stack)

2. Stub marshals parameters to a network message

Client machine

Client process
  \( k = add(3, 5) \)

Client stub (RPC library)

\texttt{proc: add | int: 3 | int: 5}

Client OS
A DAY IN THE LIFE OF AN RPC

2. Stub marshals parameters to a network message

3. **OS sends a network message to the server**

![Diagram showing the process of sending a network message from client to server](image)

4. **Server OS receives message, sends it up to stub**

![Diagram showing the process of receiving and processing the network message](image)
A DAY IN THE LIFE OF AN RPC

4. Server OS receives message, sends it up to stub

5. Server stub unmarshals params, calls server function

Client machine
Client process
k = add(3, 5)
Client stub (RPC library)
Client OS

Server machine
Server process
Implementation of add
Server stub (RPC library)
proc: add | int: 3 | int: 5
Server OS

A DAY IN THE LIFE OF AN RPC

5. Server stub unmarshals params, calls server function

6. Server function runs, returns a value

Client machine
Client process
k = add(3, 5)
Client stub (RPC library)
Client OS

Server machine
Server process
8 ← add(3, 5)
Server stub (RPC library)
Server OS
A DAY IN THE LIFE OF AN RPC

6. Server function runs, returns a value

7. **Server stub marshals the return value, sends msg**

8. **Server OS sends the reply back across the network**
A DAY IN THE LIFE OF AN RPC

8. Server OS sends the reply back across the network

9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client
THE SERVER STUB IS REALLY TWO PARTS

- **Dispatcher**
  - Receives a client’s RPC request
  - **Identifies** appropriate server-side method to invoke

- **Skeleton**
  - **Unmarshals** parameters to server-native types
  - **Calls** the local server procedure
  - **Marshals** the response, sends it back to the dispatcher

- **All this is hidden from the programmer**
  - Dispatcher and skeleton may be integrated
    - Depends on implementation
Outline

1. RPC fundamentals
2. Handling failures in RPCs

WHAT COULD POSSIBLY GO WRONG?
WHAT COULD POSSIBLY GO WRONG?

1. Client may **crash and reboot**

2. Packets may be **dropped**
   - Some individual **packet loss** in the Internet
   - **Broken routing** results in many lost packets

3. Server may **crash and reboot**

4. Network or server might just be **very slow**

   All these may **look the same** to the client…

FAILURES, FROM CLIENT’S PERSPECTIVE

[Diagram showing a client and a server with a request and a reply, both marked as failed at different points in time with an 'X'.]

The cause of the failure is **hidden** from the client!
AT-LEAST-ONCE SCHEME

• **Simplest** scheme for handling failures

1. Client stub **waits for a response**, for a while
   • Response takes the form of an **acknowledgement** message from the server stub

2. If no response arrives after a fixed **timeout** time period, then client stub **re-sends the request**
   • Repeat the above a few times
   • *Still no response?* Return an error to the application

AT-LEAST-ONCE AND SIDE EFFECTS

• Client sends a “debit $10 from bank account” RPC
AT-LEAST-ONCE AND WRITES

• put(x, value), then get(x): expect answer to be value

Consider a client storing key-value pairs in a database

• put(x, value), then get(x): expect answer to be value
SO IS AT-LEAST-ONCE EVER OKAY?

- **Yes**: If they are read-only operations with no side effects
  - *e.g.*, read a key’s value in a database
- **Yes**: If the application has its own functionality to cope with duplication and reordering
  - You will implement this in Project 2 and 3

AT-MOST-ONCE SCHEME

- **Idea**: server RPC code detects duplicate requests
  - Returns previous reply **instead of re-running handler**
- **How to detect a duplicate request?**
  - **Test**: Server sees same function, same arguments twice
    - **No!** Sometimes applications **legitimately** submit the same function with same augments, twice in a row
AT-MOST-ONCE SCHEME

• **How to detect a duplicate request?**
  
  • Client includes unique **`transaction ID (xid)`** with each one of its RPC requests
  
  • Client uses **same xid** for retransmitted requests

```python
At-Most-Once Server
if seen[xid]:
    retval = old[xid]
else:
    retval = handler()
    old[xid] = retval
    seen[xid] = true
return retval
```

AT MOST ONCE: ENSURING UNIQUE XIDS

• **How to ensure that the xid is unique?**

  1. Combine a unique client ID (*e.g.*, IP address) with the current time of day
  
  2. Combine unique client ID with a sequence number
  
    • Suppose the client crashes and restarts. *Can it reuse the same client ID?*

  3. Big random number
AT-MOST-ONCE: DISCARDING SERVER STATE

- **Problem:** seen and old arrays will grow without bound
- **Observation:** By construction, when the client gets a response to a particular xid, it will never re-send it
- Client could tell server “I’m done with xid x – delete it”
  - Have to tell the server about each and every retired xid
    - Could piggyback on subsequent requests

**Significant overhead** if many RPCs are in flight, in parallel

AT-MOST-ONCE: DISCARDING SERVER STATE

- **Problem:** seen and old arrays will grow without bound
- Suppose xid = ⟨unique client id, sequence no.⟩
  - e.g. ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩
- Client includes “seen all replies ≤ X” with every RPC
  - Much like TCP sequence numbers, acks
  - How does the client know that the server received the information about retired RPCs?
    - Each one of these is cumulative: later seen messages subsume earlier ones
**AT-MOST-ONCE: CONCURRENT REQUESTS**

- **Problem:** How to handle a duplicate request while the original is still executing?
  - Server doesn’t know reply yet. Also, we don’t want to run the procedure twice
- **Idea:** Add a `pending` flag per executing RPC
  - Server waits for the procedure to finish, or ignores

**AT MOST ONCE: SERVER CRASH AND RESTART**

- **Problem:** Server may crash and restart
- **Does server need to write its tables to disk?**
- **Yes!** On `server crash and restart`:
  - If `old[]`, `seen[]` tables are only in memory:
    - Server will forget, accept duplicate requests
RPC SEMANTICS

<table>
<thead>
<tr>
<th>Retry Request</th>
<th>Duplicate Filtering</th>
<th>Retransmit Response</th>
<th>RPC Call Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>Maybe</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute Procedure</td>
<td>At-least once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit reply</td>
<td>At-most once</td>
</tr>
</tbody>
</table>

SUMMARY: RPC

- RPC everywhere!
- **Necessary** issues surrounding machine heterogeneity
- **Subtle** issues around handling failures