CSE 124: RPCS AND GRPC

George Porter
Oct 18, 2017 and Oct 20, 2017
ATRIBUTION

• 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)
int optval = 1;

/* enable sockets to be immediately reused */
if (setsockopt(serv_sock, SOL_SOCKET,
       SO_REUSEADDR, &optval,
       sizeof(optval)) != 0)
{
    die_system("setsockopt() failed");
}
PER-SOCKET TIMEOUTS

• Previous lecture covered per-process timeout via OS signal delivery

• Can enable send/recv timeouts on per-socket basis:

```c
struct timeval timeout;
timeout.tv_sec = 10;
timeout.tv_usec = 0;

if (setsockopt (sockfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout, sizeof(timeout)) < 0) error("setsockopt failed\n");

if (setsockopt (sockfd, SOL_SOCKET, SO_SNDTIMEO, (char *)&timeout, sizeof(timeout)) < 0) error("setsockopt failed\n");
```

• send/recv return “partial result”, or EWOULDBLOCK
Each thread blocks on accept(), OS chooses one to “win” for each incoming conn.
Outline

1. RPC fundamentals
2. Handling failures in RPCs
3. gRPC: Google RPC overview
4. ATM Server demo
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**

\[
X = 3 \times 10;
\]

print(X)

> 30

**Remote computing**

\[
\text{server} = \text{connectToServer}(S);
\]

Try:

\[
X = \text{server}\.\text{mult}(3,10);
\]

print(X)

Except `e`:

\[
\text{print} \; \text{“Error!”}
\]

> 30

or

> 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 ≈ 10 cycles ≈ 3 ns
  • RPC in a data center takes ≈ 10 μs (10³× slower)
    • In the wide area, typically 10⁶× 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
• 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”

```
0000 0101
0000 0000
0000 0000
0000 0000
```

```
0x1000:
0x1001:
0x1002:
0x1003:
```

```
0000 0000
0000 0000
0000 0000
0000 0101
0000 0000
0000 0000
0000 0000
```

```
int 5 at address 0x1000:
```

```
int 5 at address 0x1000:
```
**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

\[ k = \text{add}(3, 5) \]

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**
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

Client machine

Client process
k = add(3, 5)

Client stub (RPC library)

Client OS
proc: add | int: 3 | int: 5

Server machine

Server OS
A DAY IN THE LIFE OF AN RPC

3. OS sends a network message to the server

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

Client machine
- Client process
  \( k = \text{add}(3, 5) \)
- Client stub (RPC library)
- Client OS

Server machine
- Server stub (RPC library)
- Server OS
  \[ \text{proc: add | int: 3 | int: 5} \]
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**
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 \leftarrow 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
A DAY IN THE LIFE OF AN RPC

7. Server stub marshals the return value, sends msg

8. Server OS sends the reply back across the network

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

Result | int: 8
8. Server OS sends the reply back across the network

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

10. **Client stub unmarshals return value, returns to client**

![Diagram showing the process of a client-side stub marshaling data and passing it back to the client.](image)
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
3. gRPC: Google RPC overview
4. ATM Server demo
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…
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
AT-LEAST-ONECE AND WRITES

- 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
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

**At-Most-Once Server**
```python
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
Problem: seen and old arrays will grow without bound

Suppose xid = \langle\text{unique client id, sequence no.}\rangle
- e.g. \langle42, 1000\rangle, \langle42, 1001\rangle, \langle42, 1002\rangle

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
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**
SUMMARY: RPC

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

1. RPC fundamentals
2. Handling failures in RPCs
3. gRPC: Google RPC overview
4. ATM Server demo
GOOGLE RPC (GRPC)

- Cross-platform RPC toolkit developed by Google
- Languages:
  - C++, Java, Python, Go, Ruby, C#, Node.js, Android, Obj-C, PHP
- Defines services
- Collection of RPC calls

```java
service Search {
  rpc searchWeb(SearchRequest) returns (SearchResult) {} 
}
```
IDL: INTERFACE DEFINITION LANGUAGE

- Language-neutral way of specifying:
  - Data structures (called Messages)
  - Services, consisting of procedures/methods
- Stub compiler
  - Compiles IDL into Python, Java, etc.

IDL: INTERFACE DEFINITION LANGUAGE

$ protoc search.proto

Python

Java

$ protoc search.proto

$ protoc search.proto
IDL LANGUAGE: PROTOCOL BUFFERS

• Defines Messages (i.e., data structures)

```protobuf
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

We’re using version 3 of protocol buffers in 124

Name of the message

Field 1: query
Type: String

Field 2: page_number
Type: 32-bit signed int

Field 3: results_per_page
Type: 32-bit signed int
• Why do we label the fields with numbers?
• So we can change “signature” of the message later and still be compatible with legacy code.
MAKING SERVICES **EVALUABLE**

- No way to “stop everything” and upgrade
- Clients/servers/services must co-exist
- For newly added fields, old services use defaults:
  - String: “”
  - bytes: []
  - bools: false
  - numeric: 0
  - ...

# Protocol Buffers: Base Types

<table>
<thead>
<tr>
<th>Protobuf IDL</th>
<th>Python</th>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>double, float</td>
<td>float, float</td>
<td>double, float</td>
<td>double, float</td>
</tr>
<tr>
<td>int32, int64</td>
<td>int, int/long</td>
<td>int, long</td>
<td>int32, int64</td>
</tr>
<tr>
<td>uint32, uint64</td>
<td>int, int/long</td>
<td>int, long</td>
<td>uint32, uint64</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>bytes</td>
<td>str</td>
<td>ByteString</td>
<td>string</td>
</tr>
</tbody>
</table>
PROTOCOL BUFFERS: MAP TYPE

- map<key_type, value_type> map_field = N;

Example:
- map<string, Project> projects = 3;
IMPLEMENTING IN DIFFERENT LANGUAGES

IDL

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}

C++: reading from a file

Person john;
fstream input(argv[1],
    ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();

Java: writing to a file

Person john = Person.newBuilder()
  .setId(1234)
  .setName("John Doe")
  .setEmail("jdoe@example.com")
  .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
A C++ EXAMPLE

```cpp
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);

fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

- Can read/write protobuf Message objects to files/stream/raw sockets
- In particular, gRPC service RPCs
  - Take Message as argument, return Message as response
The server can throw an Exception, which is translated into an Exception in the client

- Catch try...catch and handle as appropriate

Example to follow
Outline

1. RPC fundamentals
2. Handling failures in RPCs
3. gRPC: Google RPC overview
4. ATM Server demo
SIMPLE ATM SERVER

Operations:
- login
- Account number + PIN
- deposit
- $$$
- getBalance
- logout
Simple ATM Server

- Keeping track of account + pin with “login tokens”
- After logging in, get a token
- Use token to deposit money, withdraw, transfer, ...
STARTER CODE STRUCTURE
UC San Diego