RPC

Goal

- Make the process of executing code on a remote machine as simple and straight-forward as calling a local functions.

RPC vs REST

	REST	RPC
Definition	REST is an architecture style. It exposes data as resources and CRUD operations could be used to access resources. HTTP is an implement conforming to REST styles	Exposes action-based API methods
HTTP verbs	REST will use HTTP methods such as GET, POST, PUT, DELETE, OPTIONS and, hopefully, PATCH to provide semantic meaning for the	RPC uses only GET and POST, with GET being used to fetch information and POST

	intention of the action being taken.	being used for everything else.
Serilization protocol	readablee text(XML, JSon)	ProtoBuf, Thrift
Transmission protocol	HTTP/HTTP2	Usually UDP/TCP (There are also exception. e.g. gRPC uses HTTP2)
Examples	Dubbo, Motan, Tars, gRPC, Thrift	SpringMVC/Boot, Jax-rs, drop wizard
User friendly	Easy to debug because request/response are readable	Hard to debug because request/response are not readable
Design challenges	1. Fetching multiple resources in a single request 2. The challenge of mapping operations to HTTP verbs	Hard to discover because there is limited standardization. Without a nice documentation, you won't know how to start neither what to call.

Components

Overview

- The steps are as follows:
    i. Programmer writes an interface description in the IDL (Mechanism to pass procedure parameters and return values in a machine-independent way)
    ii. Programmer then runs an IDL compiler which generates
        - Code to marshal native data types into machien independent byte streams
        - Client/server stub: Forwards local procedure call as request to server / Dispatches RPC to its implementation

Interface definition language

```
// An example of an interface for generating stubs
service FacebookService {
  // Returns a descriptive name of the service
  string getName(),

  // Returns the version of the service
  string getVersion(),

  // Gets an option
  string getOption(1: string key),

  // Gets all options
  map<string, string> getOptions()
}
```

Marshal/Unmarshal

- Challenges: For a remote procedure call, a remote machine may:
    - Different sizes of integers and other types
    - Different byte ordering
    - Different floating point representations
    - Different character sets
    - Different alignment requirements
- Challenges: Reference variables
    - What we pass is the value of the pointer, instead of the pointer itself. A local pointer, pointing to this value is created on the server side (Copy-in). When the server procedure returns, the modified 'value' is returned, and is copied back to the address from where it was taken (Copy-out).

    - This approach is not foolproof. The procedure 'myfunction()' resides on the server machine. If the program executes on a single machine then we must expect the output to be '4'. But when run in the client-server model we get '3'. Why ? Because 'x, and 'y' point to different memory locations with the same value. Each then increments its own copy and the incremented value is returned. Thus '3' is passed back and not '4'.

```
#include <studio.h>

void myfunction(int *x, int *y)
{
        *x += 1;
        *y += 1;
}
```

Server processing model

- BIO: Server creates a new thread to handle to handle each new coming request.
    - Applicable for scenarios where there are not too many concurrent connections
- NIO: The server uses IO multiplexing to process new coming request.
    - Applicable for scenarios where there are many concurrent connections and the request processing is lightweight.
- AIO: The client initiates an IO operation and immediately returns. The server will notify the client when the processing is done.
    - Applicable for scenarios where there are many concurrent connections and the request processing is heavyweight.
- Reactor model: A main thread is responsible for all request connection operation. Then working threads will process further jobs.

Binding

- Binding: How does the client know who to call, and where the service resides?

    - The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

- Where to locate host and correct server process

    - Solution1: Maintain a centralized DB that can locate a host that provides a particular service
        a. Challenge: Who administers this
        b. Challenge: What is the scope of administration
        c. Challenge: What if the same services run on different machines
    - Solution2: A server on each host maintains a DB of locally provided services
    - Please see Service discovery

Transport protocol

- If performance is preferred, then UDP protocol should be adopted.
- If reliability is needed, then TCP protocol should be adopted.
    - If the connection is a service to service, then long connection is preferred than short connection.

Serialization protocol

- Factors to consider:
    - Support data types: Some serialization framework such as Hessian 2.0 even support complicated data structures such as Map and List.
    - Cross-language support
    - Performance: The compression rate and the speed for serialization.

Semantics of RPC

At least once

- Def: For every request message that the client sends, at least one copy of that message is delivered to the server. The client stub keeps retrying until it gets an ack. This is applicable for idempotent operations.

Exactly once

- Def: For every request message that the client sends, exactly one copy of that message is delivered to the server.
- But this goal is extremely hard to build. For example, in case of a server crash, the server stub call and server business logic could happen not in an atomic manner.

At most once

- Def: For every request message that the client sends, at most one copy of that message is delivered to the server.

Designs

1. How to detect a duplicate request?
    - Client includes unique transaction ID with each one of its RPC requests
    - Client uses the same xid for retransmitted requests
2. How to avoid false detection?
    - One of the recurrent challenges in RPC is dealing with unexpected responses, and we see this with message IDs. For example, consider the following pathological (but realistic) situation. A client machine sends a request message with a message ID of 0, then crashes and reboots, and then sends an unrelated request message, also with a message ID of 0. The server may not have been aware that the client crashed and rebooted and, upon seeing a request message with a message ID of 0, acknowledges it and discards it as a duplicate. The client never gets a response to the request.
    - One way to eliminate this problem is to use a boot ID. A machine's boot ID is a number that is incremented each time the machine reboots; this number is read from nonvolatile storage (e.g., a disk or flash drive), incremented, and written back to the storage device during the machine's start-up procedure. This number is then put in every message sent by that host. If a message is received with an old message ID but a new boot ID, it is recognized as a new message. In effect, the message ID and boot ID combine to form a unique ID for each transaction.

3. How to ensure that the xid is unique?
    - Combine a unique client ID (e.g. IP address) with the current time of the day
    - Combine a unique client ID with a sequence number
    - Combine a unique client ID with a boot ID
    - Big random number
4. seen and old arrays will grow without bound
    - Client could tell server "I'm done with xid x - delete it".
    - Client includes "seen all replies <= X" with every RPC
5. Server may crash and restart
    - If old[], seen[] tables are only in meory, then the user needs to retry

```
if seen[xid]:
        retval = old[xid]
else:
        retval = handler()
        old[xid] = retval
        seen[xid] = true
return retval
```

Last of many

- Last of many : This a version of 'At least once', where the client stub uses a different transaction identifier in each retransmission. Now the result returned is guaranteed to be the result of the final operation, not the earlier ones. So it will be possible for the client stub to tell which reply belongs to which request and thus filter out all but the last one.

Implementations

History

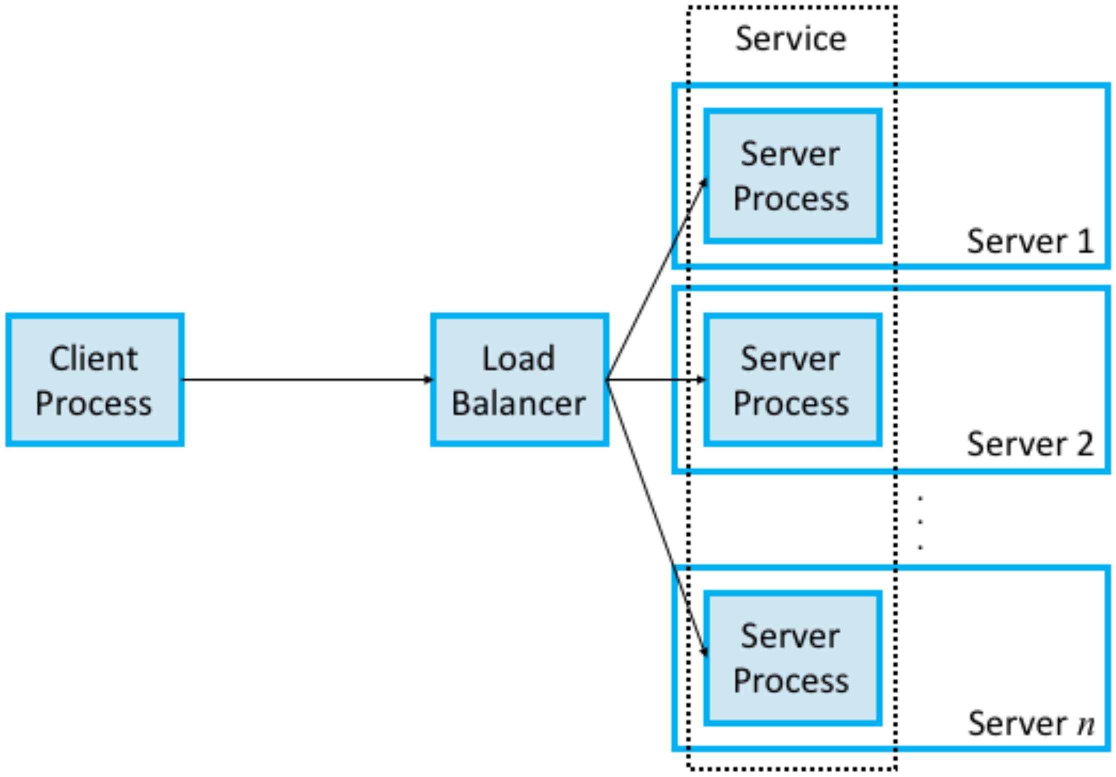
- SunRPC is the basis for Network File System.
- DCE-RPC is the basis of Microsoft's DCOM and ActiveX.
- RMI
    - RMI uses Java Remote Messaging Protocol for communication. It has limitation that both the sender and receiver need to be Java programs.

It could not be used in cross-language scenarios
    - RMI uses Java's native approach for serialization and deserialization. The generated binary format is not efficient.

gRPC

History

- The biggest differences between gRPC and SunRPC/DCE-RPC/RMI is that gRPC is designed for cloud services rather than the simpler client/server paradigm. In the client/server world, one server process is presumed to be enough to serve calls from all the client processes that might call it. With cloud services, the client invokes a method on a service, which in order to support calls from arbitrarily many clients at the same time, is implemented by a scalable number of server processes, each potentially running on a different server machine.
- The caller identifies the service it wants to invoke, and a load balancer directs that invocation to one of the many available server processes (containers) that implement that service



Features

Multi-language, multi-platform framework

- Native implementations in C, Java, and Go
- Platforms supported: Linux, Android, iOS, MacOS, Windows

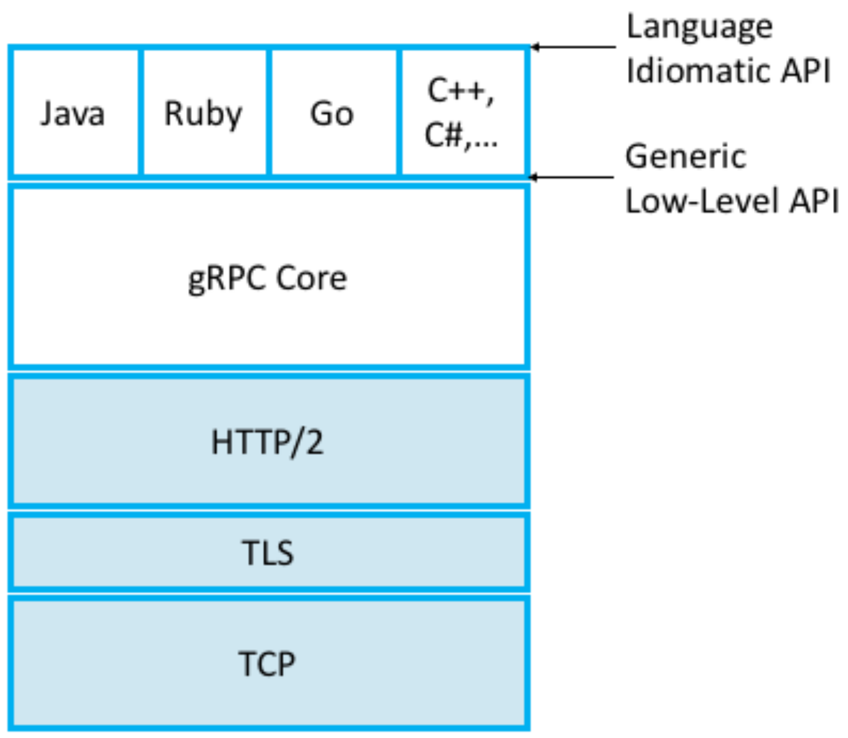
Transport over HTTP/2 + TLS

- First, gRPC runs on top of TCP instead of UDP, which means it outsources the problems of connection management and reliably transmitting request and reply messages of arbitrary size.
- Second, gRPC actually runs on top of a secured version of TCP called Transport Layer Security (TLS)—a thin layer that sits above TCP in the protocol stack—which means it outsources responsibility for securing the communication channel so adversaries can't eavesdrop or hijack the message exchange.
- Third, gRPC actually, actually runs on top of HTTP/2 (which is itself layered on top of TCP and TLS), meaning gRPC outsources yet two other problems:
  - Binary framing and compression: Efficiently encoding/compressing binary data into a message.
  - Multiplexing: Requests by introducing concept of streams.
    - HTTP: The client could send a single request message and the server responds with a single reply message.
    - HTTP 1.1: The client could send multiple requests without waiting for the response. However, the server is still required to send the responses in the order of incoming requests. So Http 1.1 remained a FIFO queue and suffered from requests getting blocked on high latency requests in the front Head-of-line blocking
    - HTTP2 introduces fully asynchronous, multiplexing of requests by introducing concept of streams. lient and servers can both initiate multiple streams on a single underlying TCP connection. Yes, even the server can initiate a stream for transferring data which it anticipates will be required by the client. For e.g. when client request a web page, in addition to sending theHTML content the server can initiate a separate stream to transfer images or videos, that it knows will be required to render the full page.

C/C++ implementation goals

* High throughput and scalability, low latency
* Minimal external dependencies

Components



Comparison

Cross language RPC: gRPC vs Thrift

* gRPC uses HTTP/2, serialization uses ProtoBuf
* Thrift support multiple modes:
    * Serialization: Binary, compact, Json, multiplexed
    * Transmission: Socket, Framed, File, Memory
    * Server processing model: Simple, Thread Pool, Non-blocking

Same language RPC: Tars vs Dubbo vs Motan vs Spring Cloud

* C++: Tars
* Java:
    * Spring cloud provides many other functionalities such as service registration, load balancing, circuit breaker.
        * HTTP protocol
    * Motan/Dubbo is only RPC protocol