SOC(5) Supplement for RPC

1. What is gRPC

gRPC is a remote procedure call (RPC) framework that is released by Google and makes HTTP-based remote function calls.

RPC is a protocol that is used to invoke a service on a remote server on the network. It is defined as an Interface Definition Language (IDL) interface, which is then called from the programming language through the corresponding Skeleton and Stub code.

In recent years, SOAP and RESTful using HTTP have been widely used, and RPC is rarely used. However, due to the disadvantage that the protocol for request / response is not explicit, Prememics adopting RPC method have started to come out.

Since RPC generally needs to know the request parameter and response parameter, it is necessary to define both interface conventions in IDL language and then generate code with a form that can be called by the programming language. This is called skeleton and stub code do.

As REST based on HTTP is popular, RPC disappears. On the other hand, REST has a lot of errors because it does not explicitly call parameter and response value (do not depend on API specification, check type, etc.) . Because JSON specification is transmitted through HTTP, it has a disadvantage that the speed is slow

So in recent years, the RPC concept has become popular again, and Facebook has released a binary-based RPC framework called Thrift. Google has provided PB (Protocol Buffer) as a framework for serializing messages (JSON, etc.) without supporting RPC itself. With the concept of GRPC, RPC framework was released with HTTP2 attached to Serizlaizer based on PB .

2. Feature of gRPC

Overcome limit of one-way of REST, support streaming based on HTTP2, fast performance compared to REST

The coding amount is also as low as Spring Boot, it is competitive, and it is advantageous to develop polygraph type by supporting multiple languages at the same time, but conversely, due to the limitation of native protocol, data format conversion is not free at runtime, It is difficult to introduce API GATEWAY, which opens messages in the middle to perform routing and mediation.

gRPC is a RPC framework that can be used not only between servers but also in client applications and mobile apps as it supports various languages such as Node.js, Python, Ruby, PHP, Go, Android based, Objective-C as well as Java and C / C ++.

3. gRPC Protocol Interface (Server – Client)

Developers using gRPC typically start with the description of an RPC service (a collection of methods), and generate client and server side interfaces which they use on the client-side and implement on the server side.

By default, gRPC uses Protocol Buffers as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages. It is possible to use other alternatives if desired.

Starting from an interface definition in a .proto file, gRPC provides Protocol Compiler plugins that generate Client- and Server-side APIs. gRPC users typically call into these APIs on the Client side and implement the corresponding API on the server side.

A gRPC RPC comprises of a bidirectional stream of messages, initiated by the client. In the client-to-server direction, this stream begins with a mandatory Call Header, followed by optional Initial-Metadata, followed by zero or more Payload Messages. The server-to-client direction contains an optional Initial-Metadata, followed by zero or more Payload Messagesterminated with a mandatory Status and optional Status-Metadata (a.k.a.,Trailing-Metadata).

The abstract protocol defined above is implemented over HTTP/2. gRPC bidirectional streams are mapped to HTTP/2 streams. The contents of Call Header and Initial Metadata are sent as HTTP/2 headers and subject to HPACK compression. Payload Messages are serialized into a byte stream of length prefixed gRPC frames which are then fragmented into HTTP/2 frames at the sender and reassembled at the receiver. Status and Trailing-Metadata are sent as HTTP/2 trailing headers (a.k.a., trailers).

gRPC inherits the flow control mechanisms in HTTP/2 and uses them to enable fine-grained control of the amount of memory used for buffering in-flight messages.

4. gRPC Environment Setup

To test gRPC based on golang, golang must be installed. gRPC uses protocol buffers to serialize structured objects to and from serial data exchanged between function calls.

1.1 Golang Installation

Install the downloaded compressed file with the following command.

```
tar -C /usr/local -xzf <Downloaded file>
```

After decompressing, create the directory where the go source will be located in the appropriate location and set the following environment variable in $ HOME / .profile.

```
Create directory 'work' in $HOME



mkdir $HOME/work

export GOROOT=/usr/local/go

export GOPATH=$HOME/work

export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

After completing the installation up to this point, the following code will test normal operation. First, create the following directory:

```
mkdir -p $GOPATH/src/hello
```

Then generate the following code with the file name hello.go.

```
package main



import "fmt"
```

```
func main() {

    fmt.Printf("hello, world\n")

}
```

Build up source code

```
cd $GOPATH/src/hello

go build
```

After the compilation is completed properly, execute the following command to check the result.

```
./hello

hello, world
```

1.2 gRPC Installation

Install gRPC through the following command.

```
go get google.golang.org/grpc
```

1.3 Install Protocol Buffers

Accept Protocol Buffers archive in the following path: https://github.com/google/protobuf/releases - The compressed filename you need to download is protoc- <version> - <platform> .zip.

Unpack the downloaded zip file to the appropriate location and add the corresponding location to your PATH. The following is an example of my settings.

```
#### ProtoBuf ###
export PATH=$PATH:/Users/mjkong/Dev/sdk/protoc-3.2.0rc2-osx-x86_64/bin
```

Install the Golang based protoc plugin.

```
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
```

5. Execution of Example

If you installed gRPC in 1.2, there are examples in its subdirectories. Let's take a look at helloworld. Its location is shown below.

```
cd $GOPATH/src/google.golang.org/grpc/helloworld
```

Under the above directory, you will see three directories: greeter_client, greeter_server, and helloworld. Among them, there is a .proto file under helloworld, and this file is where the gRPC service is specified. The protoc command creates a .pb.go file based on the existing file, which creates a message type to be passed to the client and server code when calling the function. Build and run the server code. The location should be run from "$ GOPATH / src / google.golang.org / grpc / helloworld".

```
go run greeter_server/main.go
```

Then open another terminal and run the client with the following command:

```
go run greeter_client/main.go
```

Execution result When "Greeting: Hello world" message appears, it is executed properly.

gRPC Service Update

The previous command was a call to a function called SayHello by the client, and then the message was received by the server and output by the client. Let's add a function called SayHelloAgain here. Update the .proto file as follows:

helloworld.proto

```
$GOPATH/src/google.golang.org/grpc/examples/helloworld/helloworld
/helloworld.proto

// The greeting service definition.

service Greeter {

  // Sends a greeting

  rpc SayHello (HelloRequest) returns (HelloReply) {}

  // Sends another greeting

  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}

}



// The request message containing the user's name.

message HelloRequest {

  string name = 1;

}



// The response message containing the greetings

message HelloReply {

  string message = 1;

}
```

Create gRPC Code

```
protoc -I helloworld/ helloworld/helloworld.proto --
go_out=plugins=grpc:helloworld
```

After modifying the server and client code, run it to verify that the modifications are reflected.

- Server – greeter_server/main.go

```
func (s *server) SayHelloAgain(ctx context.Context, in
*pb.HelloRequest) (*pb.HelloReply, error) {

    return &pb.HelloReply{Message: "Hello again " + in.Name}, nil

}
```

- Client – greeter_client/main.go

```
r, err = c.SayHelloAgain(context.Background(),
&pb.HelloRequest{Name: name})

if err != nil {

    log.Fatalf("could not greet: %v", err)

}

log.Printf("Greeting: %s", r.Message)
```

Execution

- Server

```
go run greeter_server/main.go
```

- Client

```
go run greeter_client/main.go
```

Check the execution results as follows.

```
mjmac:helloworld mjkong$ go run greeter_client/main.go
2017/03/26 22:22:28 Greeting: Hello world
2017/03/26 22:22:28 Greeting: Hello again world
```

6. Connection management and error handling in gRPC

(https://github.com/grpc/grpc/blob/a461f0d46ccf03bc8e107a7038296e237089d179/src/core/lib/iomgr/tcp_server_windows.c)

- Socket Creation

```
                                const grpc_resolved_address *addr,
                                int *port) {
  grpc_tcp_listener *sp = NULL;
  SOCKET sock;
  grpc_resolved_address addr6_v4mapped;
  grpc_resolved_address wildcard;
  grpc_resolved_address *allocated_addr = NULL;
  grpc_resolved_address sockname_temp;
  unsigned port_index = 0;
  grpc_error *error = GRPC_ERROR_NONE;


  if (s->tail != NULL) {
    port_index = s->tail->port_index + 1;
  }


  /* Check if this is a wildcard port, and if so, try to keep the port the same
     as some previously created listener. */
  if (grpc_sockaddr_get_port(addr) == 0) {
    for (sp = s->head; sp; sp = sp->next) {
      int sockname_temp_len = sizeof(struct sockaddr_storage);
      if (0 == getsockname(sp->socket->socket,
                           (struct sockaddr *)sockname_temp.addr,
                           &sockname_temp_len)) {
        sockname_temp.len = (size_t)sockname_temp_len;
        *port = grpc_sockaddr_get_port(&sockname_temp);
        if (*port > 0) {
          allocated_addr = gpr_malloc(sizeof(grpc_resolved_address));
          memcpy(allocated_addr, addr, sizeof(grpc_resolved_address));
          grpc_sockaddr_set_port(allocated_addr, *port);
          addr = allocated_addr;
          break;
        }
      }
    }
  }


  if (grpc_sockaddr_to_v4mapped(addr, &addr6_v4mapped)) {
    addr = &addr6_v4mapped;
  }
```

```
  /* Treat :: or 0.0.0.0 as a family-agnostic wildcard. */
  if (grpc_sockaddr_is_wildcard(addr, port)) {
    grpc_sockaddr_make_wildcard6(*port, &wildcard);


    addr = &wildcard;
  }


  sock = WSASocket(AF_INET6, SOCK_STREAM, IPPROTO_TCP, NULL, 0,
                   WSA_FLAG_OVERLAPPED);
  if (sock == INVALID_SOCKET) {
    error = GRPC_WSA_ERROR(WSAGetLastError(), "WSASocket");
    goto done;
  }


  error = add_socket_to_server(s, sock, addr, port_index, &sp);


done:
  gpr_free(allocated_addr);


  if (error != GRPC_ERROR_NONE) {
    grpc_error *error_out = GRPC_ERROR_CREATE_REFERENCING_FROM_STATIC_STRING(
        "Failed to add port to server", &error, 1);
    GRPC_ERROR_UNREF(error);
    error = error_out;
    *port = -1;
  } else {
    GPR_ASSERT(sp != NULL);
    *port = sp->port;
  }
  return error;
}


static grpc_error *prepare_socket(SOCKET sock,
                                  const grpc_resolved_address *addr,
                                  int *port) {
  grpc_resolved_address sockname_temp;
  grpc_error *error = GRPC_ERROR_NONE;
```

```
  error = grpc_tcp_prepare_socket(sock);
  if (error != GRPC_ERROR_NONE) {
    goto failure;
  }

  if (bind(sock, (const struct sockaddr *)addr->addr, (int)addr->len) ==
      SOCKET_ERROR) {
    error = GRPC_WSA_ERROR(WSAGetLastError(), "bind");
    goto failure;
  }

  if (listen(sock, SOMAXCONN) == SOCKET_ERROR) {
    error = GRPC_WSA_ERROR(WSAGetLastError(), "listen");
    goto failure;
  }

  int sockname_temp_len = sizeof(struct sockaddr_storage);
  if (getsockname(sock, (struct sockaddr *)sockname_temp.addr,
                  &sockname_temp_len) == SOCKET_ERROR) {
    error = GRPC_WSA_ERROR(WSAGetLastError(), "getsockname");
    goto failure;
  }
  sockname_temp.len = (size_t)sockname_temp_len;

  *port = grpc_sockaddr_get_port(&sockname_temp);
  return GRPC_ERROR_NONE;

failure:
  GPR_ASSERT(error != GRPC_ERROR_NONE);
  char *tgtaddr = grpc_sockaddr_to_uri(addr);
  grpc_error_set_int(
      grpc_error_set_str(GRPC_ERROR_CREATE_REFERENCING_FROM_STATIC_STRING(
                             "Failed to prepare server socket", &error, 1),
                         GRPC_ERROR_STR_TARGET_ADDRESS,
                         grpc_slice_from_copied_string(tgtaddr)),
      GRPC_ERROR_INT_FD, (intptr_t)sock);
  gpr_free(tgtaddr);
  GRPC_ERROR_UNREF(error);
  if (sock != INVALID_SOCKET) closesocket(sock);
```

```
  return error;
}
```

➢ Define TCP Socket managed objects and server objects

```
struct grpc_tcp_listener {
/* This seemingly magic number comes from AcceptEx's documentation. each
     address buffer needs to have at least 16 more bytes at their end. */
  uint8_t addresses[(sizeof(struct sockaddr_in6) + 16) * 2];
  /* This will hold the socket for the next accept. */
  SOCKET new_socket;
  /* The listener winsocket. */
  grpc_winsocket *socket;
  /* The actual TCP port number. */
  int port;
  unsigned port_index;
  grpc_tcp_server *server;
  /* The cached AcceptEx for that port. */
  LPFN_ACCEPTEX AcceptEx;
  int shutting_down;
  int outstanding_calls;
  /* closure for socket notification of accept being ready */
  grpc_closure on_accept;
  /* linked list */
  struct grpc_tcp_listener *next;
};


/* the overall server */
struct grpc_tcp_server {
  gpr_refcount refs;
  /* Called whenever accept() succeeds on a server port. */
  grpc_tcp_server_cb on_accept_cb;
  void *on_accept_cb_arg;


  gpr_mu mu;


  /* active port count: how many ports are actually still listening */
  int active_ports;


  /* linked list of server ports */
  grpc_tcp_listener *head;
```

```
  grpc_tcp_listener *tail;


  /* List of closures passed to shutdown_starting_add(). */
  grpc_closure_list shutdown_starting;


  /* shutdown callback */
  grpc_closure *shutdown_complete;


  grpc_channel_args *channel_args;
};


/* Public function. Allocates the proper data structures to hold a
   grpc_tcp_server. */
grpc_error *grpc_tcp_server_create(grpc_exec_ctx *exec_ctx,
                                   grpc_closure *shutdown_complete,
                                   const grpc_channel_args *args,
                                   grpc_tcp_server **server) {
  grpc_tcp_server *s = gpr_malloc(sizeof(grpc_tcp_server));
  s->channel_args = grpc_channel_args_copy(args);
  gpr_ref_init(&s->refs, 1);
  gpr_mu_init(&s->mu);
  s->active_ports = 0;
  s->on_accept_cb = NULL;
  s->on_accept_cb_arg = NULL;
  s->head = NULL;
  s->tail = NULL;
  s->shutdown_starting.head = NULL;
  s->shutdown_starting.tail = NULL;
  s->shutdown_complete = shutdown_complete;
  *server = s;
  return GRPC_ERROR_NONE;
}
```

- gRPC Error management object

  (https://github.com/grpc/grpc/blob/a461f0d46ccf03bc8e107a7038296e237089d179/src/core/lib/iomgr/error.c)