Parallel and Distributed Computing
CS3006 (BDS-6A)
Lecture 23

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

27 April, 2023

Previous Lecture

- HDFS
- Yarn Framework
- MapReduce (Example of word count problem)
- RPC
  - Stubs
  - Marshalling of data

Programming With Interfaces

- The term service interface is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

Benefits:

1. The user does not need to know the implementation details. They interact with the interface only.
2. The programmers also do not need to know the programming language or underlying platform used to implement the service.
3. This approach provides natural support for software evolution in that implementations can change as long as the interface (the external view) remains the same

Interface Definition Language

- Many existing useful services are written in C++ and other languages.

- It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely. *Interface definition languages (IDLs)* are designed to allow procedures implemented in different languages to invoke one another.

- An IDL provides a *notation for defining interfaces* in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

Sun RPC compiler

- Programming languages do not support Sun RPC.
  - A separate pre-compiler, rpcgen, must be used
- Input:
  - Interface definition language
- Output:
  - server stub routine
  - client stub functions
  - header file
  - data conversion functions, if needed

RPCgen

- As input, it takes a list of remote procedures (interfaces) defined in an interface definition language (IDL).
- The output from rpcgen is a set of files that include:
  - server stub: main function that sets up a socket, registers the port with a name server, listens for and accepts connections, receives messages, *un-marshals* parameters, calls the user-written server function, *marshals* the return value, and sends back a network message.
  - client stub: code with the interface of the remote function that *marshals* parameters, sends the message to the server, and *un-marshals* the return value
  - header: contains definitions of symbols used by client and server as well as function prototypes
  - data conversion functions: a separate file may be generated if special functions need to be called to convert between local data types and their marshaled forms.

Interface Definition Language

- Used by *rpcgen to generate stub functions*
- Defines an RPC *program: collection of RPC procedures*
- Structure:

```
type definitions

program identifier {
        version version_id {
                procedure list
        } = value;
        ...
} = value;
```

```
program PROG {
        version PROG1 {
                void PROC_A(int) = 1;
        } = 1;
} = 0x3a3afeeb;
```

The case of gRPC

- Example – HelloWorld application using C++ and gRPC
- Source: https://grpc.io/docs/languages/cpp/quickstart/

The protobuf file (.proto) in gRPC

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (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;
}
```

Adding a new RPC to the .proto file

```
// 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;
}
```

Regenerate gRPC code

Before you can use the new service method, you need to recompile the updated proto file.

From the example build directory examples/cpp/helloworld/cmake/build, run:

```
$ make -j 4
```

This regenerates helloworld.pb.{h,cc} and helloworld.grpc.pb.{h,cc}, which contains the generated client and server classes, as well as classes for populating, serializing, and retrieving our request and response types.

Update and run the application

Update the server

Open `greeter_server.cc` from the example's root directory. Implement the new method like this:

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    // ...
  }

  Status SayHelloAgain(ServerContext* context, const HelloRequest* request,
                       HelloReply* reply) override {
    std::string prefix("Hello again ");
    reply->set_message(prefix + request->name());
    return Status::OK;
  }
};
```

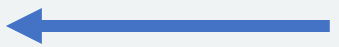
Update the client

A new `SayHelloAgain()` method is now available in the stub. We'll follow the same pattern as for the already present `SayHello()` and add a new `SayHelloAgain()` method to `GreeterClient`:

```
class GreeterClient {
 public:
  // ...
  std::string SayHello(const std::string& user) {
     // ...
  }

  std::string SayHelloAgain(const std::string& user) {
    // Follows the same pattern as SayHello.
    HelloRequest request;
    request.set_name(user);
    HelloReply reply;
    ClientContext context;

    // Here we can use the stub's newly available method we just added.
    Status status = stub_->SayHelloAgain(&context, request, &reply);
    if (status.ok()) {
      return reply.message();
    } else {
      std::cout << status.error_code() << ": " << status.error_message()
                << std::endl;
      return "RPC failed";
    }
  }
```



Finally, invoke this method in the `main()`

```
int main(int argc, char** argv) {
  // ...
  std::string reply = greeter.SayHello(user);
  std::cout << "Greeter received: " << reply << std::endl;

  reply = greeter.SayHelloAgain(user);
  std::cout << "Greeter received: " << reply << std::endl;

  return 0;
}
```

Microservices and gRPC

- Please see the first 5 or 6 slides of:
  - https://slides.com/petermalina/deck-1/fullscreen

RPC Paradigm

- Instead of accessing remote services *by sending and receiving messages*, a client invokes services by making a procedure call.
- The local procedure *hides the details of the network communication*
- The main goal of RPC is to *hide the existence of the network* from a program.
- The user doesn't first open a connection, read and write data, and then close the connection

References

- Slides of Dr. Rana Asif Rehman & Dr. Haroon Mahmood

Helpful Links on gRPC:

- https://yueying-lisa-li.org/slides/grpc/
- https://arquisoft.github.io/slides/course1920/seminars/SemEn-10-gRPC.pdf

Useful Links – Sun RPC and rpcgen

- Lecture contents and compiling instructions
  - https://www.cs.rutgers.edu/~pxk/rutgers/notes/content/ra-sunrpc.pdf
- rpcgen tutorial
  - http://docs.oracle.com/cd/E19683-01/816-1435/rpcgenpguide-21470/index.html
- SunRPC for windows
  - http://gnuwin32.sourceforge.net/packages/sunrpc.htm