ELEC 477 Distributed Systems
Assignment 1 - Remote Procedure Call

Due Feb 5, 2024, 23:99.

1. Purpose

One of the key concepts in distributed systems is the remote procedure call (RPC). In this assignment you will implement a very simple versions of a remote procedure call using the User Datagram Protocol (UDP) to demonstrate an understanding of the RPC mechanism. Real world implementations of RPC are somewhat more complicated. You will then use the RPC to build added functionality to store aircraft information.

It will also serve as an introduction to C++ and to the use of simulated network connections. One of the difficulties in programming distributed systems is dealing with failure. Failure of network segments or routers, or failures of servers. Running your programs on a single machine reduces the chance of error. The simulated network library will introduce errors into your communications to allow you to test the robustness of your solution.

2. Lab Setup

Download the network simulator from week 1. Create a new directory for assignment1. You can choose to copy all of one of the example folders and modify the files, or start fresh using the example files as a guide. The files that must be copied ***unmodified*** are: network.hpp, network.cpp, dumpHex.cpp, and dumpHex.hpp. They provide the core of the network simulator. Read the documentation of the simulator before .

Accounts are available on whisper.engineering.queensu.ca. If you are running on your own machine, you must install a C++ compiler, make, gdbm (Linux only), protobuf-compiler version 3.6.1 or later (I've only tested with 3.6.1, latest available on Ubuntu 20). protobuf-compiler 3.6.1 is only available for x86 macintosh, ARM will have to try a later version.

3. Review the Basics of IP Communication

Each network interface on a computer has an address. This assignment is based on version 4 of the internet protocol, still the most common version of the protocol in use.

Addresses in IPv4 are 32 bit numbers expressed as the decimal value of each byte separately (e.g. 192.168.0.1). Each packet originates from a port on one of the interfaces and has a destination address that consists of an address and a port. Common port numbers are TCP port 80 for unencrypted web traffic (http), and TCP port 443 for encrypted web traffic (https). TCP builds a reliable continuous connection between a port on one computer and a port on the other computer. UDP sends unreliable individual messages between ports on the same or different interfaces. This assignment will use UDP communication.

4. Remote Procedure Call

An RPC is similar to a local procedure call such as the call to library function *sprintf*. An RPC takes parameters and returns a result, however, the parameters are sent to another computer(the *server*) where the code of the procedure is executed and the result is returned to the computer (the *client*) where the call occurred.

In this assignment you are going to implement a single RPC service that implements a persistent dictionary known as a key value server. It consists of two remote procedure calls. The first RPC, *kvPut*, will store an arbitrary byte array value using an integer key on the server. The return value indicates if the value was successfully stored. The signature of the *kvPut* method is:

```
bool kvPut(int32_t key, const uint8_t * value, uint16_t vlen);
```

The second call, *kvGet*, takes an integer key value, and returns a pair of values, the first value indicates if the value is valid. It has a signature of:

```
struct kvGetResult{
    bool status;
    uint8_t * value;
    uint16_t  vlen;
};
kvGetResult kvGet(int32_t key);
```

In both cases, the return value *true* means the operation was successful. In the case of *kvPut*, this means that the value has been stored, for *kvGet*, the value was successfully retrieved. The value *false* means that the operation failed, the value could not be sent or stored(*kvPut*), or the value was not found (*kvGet*). For this assignment we will not distinguish between a failure returned by the server and a failure returned by the client stub because of a timeout. You can choose to change the API for the remote procedure call to use smart_pointers if you wish, such as

```
  bool kvPut(int32_t key, unique_ptr<uint8_t[]> &value, uint16_t
vlen);

  struct kvGetResult{
      bool status;
      unique_ptr<uint8_t[]> value;
      uint16_t  vlen;
  };
```

5. Marshalling and Unmarshalling the Data

As part of a remote procedure call, the parameters must be sent to the server and the result returned. In this assignment, we are modelling using UDP as a transport for the RPC messages. While UDP can carry messages up to just under 64K in size, in practice the maximum safe UDP message size is 508 bytes. For the purpose of this assignment, we are modelling the maximum message size as 1400 bytes, which is just under the max message size for single packets on local ethernet networks. Larger message will be fragmented (split into multiple ethernet packets). For simplicity, our network simulation does not model fragmented UDP messages, so any UDP messages over 1400 bytes will be rejected (return value -1, errno == ENOMEM).

As discussed in class, there are multiple way in which the parameters and results can be encoded to send over the network. This assignment will use binary format in network byte order.

Some, but not all, protocols include a magic number or signature as the first several bytes of a message. The magic number is a constant value used in all messages of that protocol. This makes network diagnostics easier if different ports are used for the protocol over time. You will have to choose an appropriate magic number for your implementation of the Key Value protocol.

Since messages may be lost, delayed, or even repeated, you must include a unique message identifier (i.e. a serial number). In the future, the RPC protocol may be changed, for example, adding *kvDelete* or *kvAppend* procedures. So you will need a version code.

Thus all messages in your RPC protocol must start with a message header that consists of a magic number, a version, a message unique identifier. There are a total of four message types. The request and response messages for *kvPut* and the request and response messages for *kvGet*.

You will use Google Protobuf to marshal and unmarshal the messages. Start with creating a .proto file to contain your message types. Since a single network port is used for an RPC service, the message must indicate which procedure is being invoked. This means you can only use a single message type in protobuf for the message and that it must contain variant messages (hint use the oneof keyword). Separate the header information from the variant messages by putting each of the variants in a user defined type. (i.e. the user defined types for put request, get request, put response and get response must not contain the header information).

One design decision that is up to you is if you combine everything into a single message type, or if you have separate message types for the requests sent to the server and the response sent back to the client. In general your messages should start with the header fields that are common to all messages and then the variant types of message. Use a separate user defined type for each of the request and response types to separate them from the header information.

Add or modify the Makefile to generate a .o file from the .proto file. The since the protoc command generates two output files, and example form of the Makefile command is:

```
exproto.pb.h exproto.pb.cc: exproto.proto
	protoc --cpp_out=. exproto.proto
```

Replace exproto with the name of your protobuf definition file. Note that the second line must start with a tab character. The first line says that two files with the suffix .pb.h and .pb.cc depend on your .proto file. The second line gives the command to create or update those files if the .proto file is newer than the generated files.

6. The Simulated Computers

As described in the simulation documentation each of the computers in the simulation are instances of class *Node*. While a server could also be a client of another server, for simplicity in this assignment you will create subclasses of the *Node* class, one for the server, multiple classes for the clients. Give those classes reasonable names. You can use the samples in the example folders as a guide. Your simulation will model multiple independent keyservers and clients. You should read the following sections first before you start implementing the classes. Start with stubs of each the classes and fill them in as you go along. You should compile and test run regularly to ensure you always have working code.

6.1. The Server

The constructor of your service should create an instance of your key value service (see the next section) and use the addService method to add to the list of services in the node.

Your servers will use the *gdbm* library to store the data that is sent to them by the clients. Since you will have multiple independent servers, each must know the gdbm file that will contain the data. Add a method to your server class interface that will be used to set the name of the file, this method in turn will set the name in the service class. You will need to add an instance variable to so that the file name can be forward to the service.

6.2. The Service

Create a class that is a derive class from the Service class.

Your servers will use the *gdbm* library to store the data that is sent to them by the clients. On MacOS, you can use *ndbm* instead. Since you will have multiple independent servers, each must know the gdbm file that will contain the data. Add a method to your service class interface that will be used to set the name of the file, this method will be called by the equivalent method in your server class. GDBM has some fault tolerant properties, but we will not be using them in this assignment.

The stop function should simply set the alive field to false. Your other code should check the alive field regularly (for example as a condition of the loop that waits for a message from the client). The network library will also check the alive flag of the service and throw an exception if it discovers that the alive flag is false. Do not catch this exception in your service class, it is caught in the node class to deal with the termination of a service.

The start function is the control function of your service, when it exits, then the service stops and the thread running the service exits. This code should open the *gdbm*/*ndbm* file, setup the network configuration and run the main network loop that waits for messages from clients, and processes them). After the loop it should clean up the sockets, close the file and return to allow the service thread to exit.

You should have three additional message. The methods that implement put and get have the same signature as the client signature. They should use the *gdbm/ndbm* methods to store and retrieve the value appropriately.

Your third method is responsible for checking the header information, determining which procedure was requesting and then calling the appropriate method (dispatch). A design decision for your team is where the data should be marshalled and unmarshalled. In the start method or in the dispatch method, either is ok. The point is that the start method and the dispatch method together form the server stub of the RPC call as shown in the week2 class 2 lecture. Note you will need to use the mutable methods on the classes generated by protobuf.

6.3. The Clients

Like the server, the client is a subclass of *Node*. One difference is that the start routine of the client node runs the client code. It must contain the code that implements the simulated client, and calls the client stub code that implements the interface described in section 4.

You should put the client stub methods (*kvPut,kvGet*) a second class which is instantiated by the start method. This subclass also has a function to set the address of the server. Note you will need to use the mutable methods on the classes generated by protobuf. You should manage timeouts and check the serial number in case the return message is a delayed message from a previous call.

You can use the C++ sleep_for function from the *chrono* library to space out the calls in the startup function to simulate client activity, or you can write some code that would simulate some traditional use of a persistent dictionary. Note, that all node instances run in the same process, so there is no keyboard interaction.

The separate class to implement the client stub allows you to easily create multiple clients whose difference is the contents of the start() method.

The second difference, is that instead of a method to set the database file name, you will add a method to set the server address to which the client will connect. This function will call the matching function on the client stub so that it knows which server to connect to. This will allow you to start two servers and have several clients connect to either server.

One last note is that the client stub should "autostart". That is it should not create any network connections until the first call to a stub routine is called. It should then initialize the network connection before making the call to the server. It should also have a shutdown method to close all of the network connections involved with the stub.

6.4. The Main Function

The main program is provided by main.cpp. The program takes several command line arguments that are used to control the parameters of the simulation and any debugging or tracing of simulated network activity. The main program will start a thread for each simulated computer in your program. Use the main.cpp function from one of the examples as a template.

The first part of the program starts the servers. One difference from the example code is that if you want to call the set database file name method, your shared pointer will have to have the same type as your class (not the class Node). Add that call before you call the startServices method on your server instance.

The next part of the code creates the clients. You should make a short wait between starting the servers and clients to ensure that the servers create or open their DBM files and set up the network connections. Like the server, the type of the shared_ptr will have to be your class rather than Node as shown in the examples so that you can call the set server address method. After creating the clients, setting the address of the client and server, you call the start method of each client in its own thread. The thread must be added to the nodes and names dictionary for the network library to properly track your client behaviour. Note you will have to make a C++ scope to create a lock_guard when adding the client threads to the nodes and names dictionaries.

The last part is to wait for the clients to finish. Once all clients have finished, then you can call stopServices and waitForServices on each of the servers you started in part 1 of the main function. You can then clean up with ShutdownProtobufLibrary().

7. The Simulated UDP network

The User Datagram Protocol (UDP) provides an unreliable datagram message service using Internet Protocol (IP) packets. Packets that are sent by one computer are not guaranteed to be received by the destination computer. However, when the source and destination computer are the same computer or on the same local network, as in this assignment, UDP messages are almost never lost. Since you have to write and test code that may involve the loss of UDP messages, we provide an alternate library (file network.cpp) that will simulate the connection between the client and server. This file provides a subset of the network functions provided by Linux. If you use any network library routines that have not been implemented by the simulated network library, they will invoke the real library. Since the other interface functions are bound to the simulated network library, the results will be unpredictable. There are a few functions from the standard library that can be used with the simulated library that are listed below. Other than the listed real library functions, your assignment must only use the functions that are implemented by the simulation library for network access.

7.1.Network Simulation Library Interface

The following table gives the signatures and a short description of the functions.

Signature	Description
int socket(int domain, int type, int protocol);	Returns a socket, represented by an int, in the given protocol domain (AF_INET for IPV4) of the given type (use SOCK_DGRAM or UDP) with the protocol (use the value 0). Returns -1 if an error.
int bind(int *sockfd*, const struct sockaddr **addr*, socklen_t addrlen);	Binds a socket to the address given by the second parameter. Only needed in the server.
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);	Receives a UDP message on a socket. The address of the sender will be stored in the second last parameter. Returns the number of bytes received.
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);	Sends a UDP message to the address given in the 2nd last parameter.
int setsockopt(int sockfd, int level, int optname, const void * optical, socklen_t *optlen)	Set system level parameters for a given socket.
remoteHost =gethostbyname(hostName);	Query the DNS server to translate a domain name into an internet address.

Several existing network function that do not interact with socket can be used in the assignments. These are the four functions used to translate between host and network byte order described above, and the function *inet_aton* which translates a string containing an ip address (e.g. "127.0.0.1") to a 32 bit IPv4 address.

The simulated functions use the same structures and flags as the real network library. To open a socket use the following code:

```
socketid = socket(AF_INET, SOCK_DGRAM, 0)
```

Where *socketid* is the integer that represents the socket (name it something more appropriate than *socketid*).

To bind a socket to a given address in your server code., you first fill in the fields of a variable of type *sockadd_in*. The three fields that must be given values are the sin_family (set to AF_INET), sin_addr.s_addr field (INADDR_ANY for any) and the sin_port (the port number in network byte order).

8. Socket Programming

The interface to ip networking is the socket. On Unix(i.e. Linux, Mac OS, iOS and Android) systems, the socket is implemented as a low level file descriptor with extra operations. The general algorithm for the server is:

```
1. Create a socket
2. Bind the socket to an interface and port
3. Begin Loop
4.      Receive a message (using recvfrom)
5.      Unmarshall the parameters (and metadata)
6.      Verify magic, and version
7.      Determine which function is called
8.      Cal the implementation of the function
9.      Marshal the result (and metadata)
10.     Send the reply to the sender of the received message
11.End Loop
```

Both functions use the same port and thus the same socket. Which function is performed is part of the UDP message.

The general algorithm of the client side of each of the RPC functions is:

```
1. Create a socket
2. Increment the serial number
3. Marshal the metadata and parameters
   (including which function is being called)
4. Send message to server
5. Begin Loop
6.      Receive a message (using recvfrom)
7.      Unmarshall the potential result (and metadata)
8.      Verify magic and version
```

```
9.      Verify serial number (back to 5 if incorrect)
10.     use the result
11.End Loop
```

9. Testing

You should test several scenarios, which will involve writing several clients.

1. The first test should have a simple client that puts a value and retrieves it and checks that it got the same value back.
2. You must test that data containing null bytes is stored and retrieved correctly.
3. You should have tests with two servers and two clients to show independence.
4. You should have tests that write different values to the same server with the same key to overwrite, this test should be timed so that if one of the packets is delayed, the value returned at the end by one of the clients is different.

You should be able to reuse some of the client classes in the tests.

10. What to hand in

During the assignment, git repositories on code.engineering.queensu.ca will be created for each of the groups. Once they are created, you can move your code into your repository and use it to manage your assignment. Please make a separate directory for each assignment with a reasonable name. As with ELEC 377 assignments, you are expected to provide two documents. One that describes your code and how you solved the assignment, and the other that describes your test cases and test rationale.

The design document should describe the problem and how you solved it. It should enumerate any design designs you made, such as how to split the processing in the server.

The documents should describe each of your test cases. Do not include the test results in the document. The test results should be saved in individual files. Instead, you testing document should identify the file and describe what it tests. For example: "File test1.txt contains the output of our first test. It shows that the value 'Smith Engineering is great" was stored with the key 1, and successfully retrieved using the key 1". You will need to have appropriate print statement in the client so that the output shows that the test is successful.

The marking scheme for this assignment will be out of 20.

Documentation of your solution	6
Code Correctness	6
Style And Clarity	4
Testing Documentation and Cases	4