CS 3210

Remote Procedure Call

& Lab 6 Implementation

Team 12: Ning Wang, David Benas, Zongwan Cao

Final Report

4/22/16

1. Problem & Motivation

Our initial motivation for pursuing a project on adding RPC functionality to the networking capabilities developed in Lab 6, was primarily that the default socket interface is just incredibly difficult and obtuse to use on its own. We wanted to add support for remote procedure calls to impose a layer on top of the socket interface making it easier to build distributed systems, etc. In order to accomplish this, we first had to finish the E1000 driver for JOS to even have networking capabilities to begin with. Then we moved on to design the interface between RPC client and server and the actual implementation. Finally, we wanted to build a simplified distributed key-value store using our RPC library to demonstrate the utility of our project.

2. Part I Design & Implementation

The implementation of network driver was carried out in the following steps:
We wrote a driver for a network interface card, which is from the Intel 82540EM chip, also known as E1000. The diagram on our presentation slides describes different environments of network server and their relationships. This diagram shows the entire system including the device driver. Based on lab6, we implemented the parts highlighted in green.

Initialization and transmitting packets:
(1) Making kernel time notion by *time_tick()* call.
(2) Creating a vm mapping to assign E1000 an MMIO region to store its base and size.
(3)After setting up transmit control registers and transmit descriptors struct in **e1000.h**, we allocated a circular transmit descriptor list and initialized them in **e1000.c**.
(4) In **e1000.c**, we developed *e1000_tx()* to transmit a packet by checking that the next descriptor is free, copying the packet data into the next descriptor, and updating TDT register.
(5) Adding system call to transmit packets from user space.
(6) Implementing the output helper environment to send packets in **net/output.c**.

Receiving packets and web server:
(1) Setting up the receive queue and configure the E1000 by following the process in Intel's Software Developer's Manual.
(2) Like the transmit process, we developed *e1000_rx()* to receive a packet and add it to the system call. Also we finished the input helper environment in **net/input.c**.
(3) Building a web server in **user/httpd.c** to send the contents of a file to the requesting client.

3. Part II Design & Implementation

We implemented a user-level RPC library on top of the BSD socket interface provided by the core network server environment. The architecture of our RPC library is illustrated in Figure 2 below.

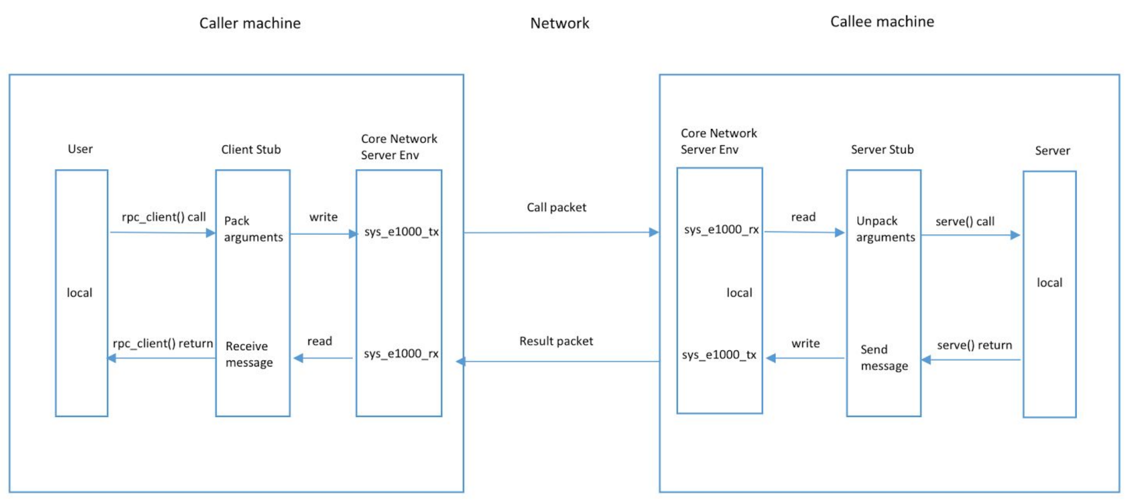

Figure 2: RPC Callee-Caller Architecture

The entire architecture is divided into caller machine and callee machine. At the caller side, the user will invoke the *rpc_client()* call with input arguments, just like any other function calls. The arguments will be marshalled into a RPC packet by the user stub code in rpcclient.h and transmitted using the socket interface provided by the core network server environment. It internally uses the input helper environment to invoke *sys_e1000_tx()* syscall to transmit the packet. At the callee side, the core network server environment will receive the packet. Then the server stub code can read the packet using socket and unpack the arguments from the RPC packets. Then it will invoke the actual serve function on behalf of the user. When the serve function returns, it will send a packet with result back to the caller machine. Then the client stub code at the caller side will get the message, copy the result into the user-provided buffer and return to the user local side. At this point, the remote procedure call finally returns.

Design of the RPC library

Structure of the RPC packet is designed as follows.

```
struct rpc_pkt {
    int num;                             // the number of arguments actually used
    int types[MAXARGS];                  // the data types for these arguments
    char data[PKTLEN -                  // the actual arguments transmitted
           -   (RPCMAXARGS + 1) * 4];
};
```

The size of a RPC packet is 1024 bytes, which is less than the maximum packet size (1518 bytes) supported by our network driver. The num and types are the information about this RPC packet. The server stub will use these information to decode the arguments from the packet.

However, the users of the RPC library shouldn't be aware of this structure to do RPC. We present several function calls as the interfaces between the RPC client and RPC server.

RPC server:

The RPC server program will include inc/**rpcserver.h** to use the library. The two functions used by rpc server are: *int rpc_server_init(uint16_t port);* and *void rpc_server(serve_function serve);*

The server program can call *rpc_server_init()* to bind to a local port and listen to incoming RPC requests. The return value indicates whether this binding is successful. The server call rpc_server to pass a function pointer which points to the server function to be executed. This function call also starts the server and begin accepting requests until it is cancelled. This also means it will run forever without being interrupted.

RPC client:

We present the following three functions for RPC client. The RPC client program will include inc/**rpcclient.h** to use the library.

int rpc_client_init(char ipaddr, uint16_t port);*

The above is to initialize the RPC session and tell the library which IP address and port to make the RPC call.

int rpc_client_args(int t0, int t1, int t2, int t3, int t4, int t5);

The above is to tell the RPC runtime the data types used for the 6 available arguments. If the users do not use an argument, they can pass NTYPE to this function.

int rpc_client(void res, int a0, int a1, int a2, int a3, int a4, int a5);*

Finally, the above is the RPC function call. Users should pass a pointer to a buffer used to receive the RPC message from the server. a0-a5 are the values for the six arguments. If the user does not use an argument, they can pass 0 to that input argument. The RPC client stub code will marshall those arguments into a single RPC packet based on the information passed in rpc_client_args. Then it will try to connect the RPC server and send the packet. Then will wait and read the results from the packet received from RPC server and copy it back to the user supplied buffer.

How arguments marshalling works

The data type information is kept as a global array *arg_types* in **rpcclient.h**. The implementation for packing arguments into RPC packet is done in arg_copy in the same file. The client stub keep using this helper function to copy the six arguments into the packet. If the type is CHAR or INT, it copies the value. If the type is STRING, it recognizes the input argument as a pointer to a string and strcpy that string into the packet. At the server side, the server stub first reads the num field from the rpc_pkt struct, then it knows the next num ints are defining the data types. Starting from pkt.data (where the actual arguments start), it copies the arguments to a temporary buffer according to their data types and use those arguments to invoke serve() function.

4. Challenges

Throughout the process, we all encountered our own difficulties and obstacles that made it harder to progress the way we had initially planned. We were able to complete the lab itself without too much difficulty, beyond some minor bugs which did not take too long to fix.

However, there were some obstacles during the implementation of the RPC portion. The most difficult part is to design the interface between client and server. At first,we hardcoded the number of arguments be to 2 and the data types to be all int. That's enough to implement the simple distributed key-value store demo, but not really generalized. So we decide to support three different data types and let the user to choose how to use the six available arguments. This also increases the difficulty to marshal the arguments into the network packet. But we managed to do it. Implementation of RPC is not easy, and testing is even harder. First we tried to set up a VLAN between multiple QEMU instances. But for some reasons the client instance was not able to connect to the server instance. So we chose the let the server forward the port to the host OS and let the client instances connect to the forwarded port on host OS. Then it worked just fine. We used this method to do the demo.

5. Evaluation

All in all, the project was an educational and fun experience. This was the first time that the lab involved us implementing portions of the project entirely from scratch, which gave a lot of flexibility to implement the E1000_tx and _rx functions however we saw fit. In turn, since we wrote most of the code from scratch, debugging was simpler as we knew how each portion of the code interacted with the rest of the driver. While at times the transcribing of default and initialization values from the x86 manual tables was a tedious task, being able to code something directly in line with specifications was a somewhat novel experience for some of us. The addition of the RPC implementation allowed us to understand the benefits of an RPC library on top of an existing socket interface and the simplicity it brings to the coding of networked applications.

References

[1] JOS Lab 6: Network Driver

(https://tc.gtisc.gatech.edu/cs3210/2016/lab/lab6.html)

[2] Intel's PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual (https://tc.gtisc.gatech.edu/cs3210/2016/r/hardware/8254x_GBe_SDM.pdf)

[3] QEMU Manual

(http://wiki.qemu.org/Manual)

[4] Birrell, Andrew D., and Bruce Jay Nelson. "Implementing remote procedure calls." ACM Transactions on Computer Systems (TOCS) 2.1 (1984): 39-59.