Riley Deal

CSCI 53700 Distributed Computing

Assignment 3

Analysis of results & Design Principles

For pros and cons, I believe that outside the benefits of learning how rpc works at a core level, rpcgen has near zero pros to go along with it. While it does build template code for the programmer to work with, the template code is near unreadable in places, and provides no help to the user on what they are doing wrong. In addition to this, any small change to the .x file requires a regeneration of the base files, unless you wish to understand where all the changes need to take place, which is worth your time just to avoid having to drop your custom code back into ten different places. Making the generation "benefit" pretty useless with even a low-level understanding.

For example, in the early parts of the project I was attempting to use int * as a return type in the *.x file. Which it simply said was not allowed. This was strange as until this point everything was basically c syntax. However, **typedef int* whatever_name** did seem to work at a surface level. Without further exploration this seemed like magic, but this was due to typedefs creating an XDR data type in the background by default. However, since I wasn't actually using this data type correctly it still did not work. Finally, after reading and understanding the requirements to correctly make the XDR data type for int * I was able to get it working. Nowhere during compilation did it indicate that this could be an issue, which made the problem deeply nested and hard to find.

The next issue I ran into were issues with memory allocation. At first I was trying to allocate memory on the client, the server, etc. Which was causing frequent segmentation faults. This was remedied by allocating space on both the client and the server. This seems obvious in retrospect, but there is so much happening in hard to understand files behind the scenes already,

that it seemed like possibly at least the result on the server side would be allocated already, leading me towards these issues.

My final issues came when attempting to add threading and concurrency. This requires several flags such as -M for multi thread safe code and -A for multi thread auto mode, which enables concurrency by default. -M changes several things which were incompatible with multi threaded code, such as static return variables in each function, however it doesn't actually add multi threading to the default code generation, interestingly enough. On top of this, adding multithreading requires editing of the *_svc.c file, which clearly states at the top ***NOT*** to edit it.

Example of the necessary intermediary function for multithreading which is added between main and add_prog_1 call in add_svc.c

```
// intermediary function in order to hand control to a detached thread
// This gives us Multithreaded server, but not concurrency. Concurrency
// Requires -A which is not available on linux
static void start_thread(struct svc_req *rqstp, register SVCXPRT *transp) {
  std::thread thread(add_prog_1, rqstp, transp);
  thread.detach();
}
```

This is all simple to add once you understand what is happening, but where things get difficult is the next part. The -A flag is not supported in any way on linux OS, meaning getting consistent concurrency by default is very difficult if not impossible. This would require digging into the core of rpcgen to understand what is going wrong. These claims are backed by the following source:
https://stackoverflow.com/questions/33202866/rpc-cant-decode-arguments-for-tcp-transport

Specifically, the user Craig Estey replies that even -M shouldn't work as it is not shown in the Linux man pages (which is true, shown here https://linux.die.net/man/1/rpcgen). The original poster replies saying that it did generate code, which is also true for our project. Craig Estey later replies, "You are missing something, but you'd have to dig really deep to find it, so don't feel bad :-) It's not what rpcgen does [which is correct]. It's that MT RPC servers aren't supported by the underlying libs in Linux. The generated code you used was correct, but still didn't work…". To me this implies that rpcgen will happily generate code ready to support MT RPC servers for you...they just won't work as intended.

rpcgen with and without -A, this shows Linux not recognizing -A

```
[rjdeal@tesla MT_TEST]$ rpcgen -A -M -N -C -a add.x
usage: rpcgen infile
       rpcgen [-abkCLNTM][-Dname[=value]] [-i size] [-I [-K seconds]] [-Y path] infile
       rpcgen [-c | -h | -l | -m | -t | -Sc | -Ss | -Sm] [-o outfile] [infile]
       rpcgen [-s nettype]* [-o outfile] [infile]
       rpcgen [-n netid]* [-o outfile] [infile]
options:
-a              generate all files, including samples
-b              backward compatibility mode (generates code for SunOS 4.1)
-c              generate XDR routines
-C              ANSI C mode
-Dname[=value]  define a symbol (same as #define)
-h              generate header file
-i size         size at which to start generating inline code
-I              generate code for inetd support in server (for SunOS 4.1)
-K seconds      server exits after K seconds of inactivity
-l              generate client side stubs
-L              server errors will be printed to syslog
-m              generate server side stubs
-M              generate MT-safe code
-n netid        generate server code that supports named netid
-N              supports multiple arguments and call-by-value
-o outfile      name of the output file
-s nettype      generate server code that supports named nettype
-Sc             generate sample client code that uses remote procedures
-Ss             generate sample server code that defines remote procedures
-Sm             generate makefile template
-t              generate RPC dispatch table
-T              generate code to support RPC dispatch tables
-Y path         directory name to find C preprocessor (cpp)

For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
[rjdeal@tesla MT_TEST]$ rpcgen -M -N -C -a add.x
[rjdeal@tesla MT_TEST]$
```

Multiple Clients

Despite all this, concurrency does seem to happen, but it appears to be unstable due to undefined behavior. I have had many runs where two clients are able to get into the same function at the same time without issues. However, I have also seen many issues with double free / corruption happening on the server side which is only present during multi-client executions. This was tested by adding a 10 second sleep to the DateAndTime function and seeing both clients trigger a print on the server side within these 10 seconds.

One interesting thing to note is that server side sleeps can help AND hurt concurrent code. I was seeing the aforementioned 10 second sleeps trigger timeout and retries from the client side, which were eventually causing memory errors for some reason. Additionally, tests were performed with 2 clients hitting the server as fast as possible in an infinite loop, which eventually resulted in memory errors as well. A final test was performed with a 10ms sleep on the server side, during which neither client failed nor were their server side memory errors after

20 minutes of execution. These results also lead me towards where Craig Estey above had hinted at, that some internal structures are missing or not supporting what they claim to.

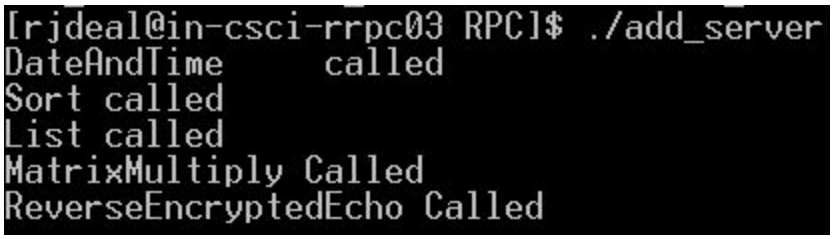
Here is an example on the server side where multiple clients calling the server appear to be causing an invalid sized free within an internal function (svc_sendreply) after many (1000s) of calls.

```
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
DateAndTime     called
*** Error in `./add_server': free(): invalid size: 0x0000000000ba2de0 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x816b9)[0x7f8daa6bf6b9]
/lib64/libc.so.6(xdr_bytes+0x42)[0x7f8daa7733a2]
/lib64/libc.so.6(xdr_accepted_reply+0x35)[0x7f8daa766335]
/lib64/libc.so.6(xdr_union+0xa7)[0x7f8daa7736c7]
/lib64/libc.so.6(+0x133c62)[0x7f8daa771c62]
/lib64/libc.so.6(svc_sendreply+0x4b)[0x7f8daa77094b]
./add_server[0x401d3c]
./add_server[0x402b0c]
./add_server[0x402a0d]
./add_server[0x4029ec]
/usr/local/stow/gcc-6.3.0/lib64/libstdc++.so.6(+0xb922f)[0x7f8daafde22f]
/lib64/libpthread.so.0(+0x7ea5)[0x7f8dab2adea5]
/lib64/libc.so.6(clone+0x6d)[0x7f8daa73c8cd]
```

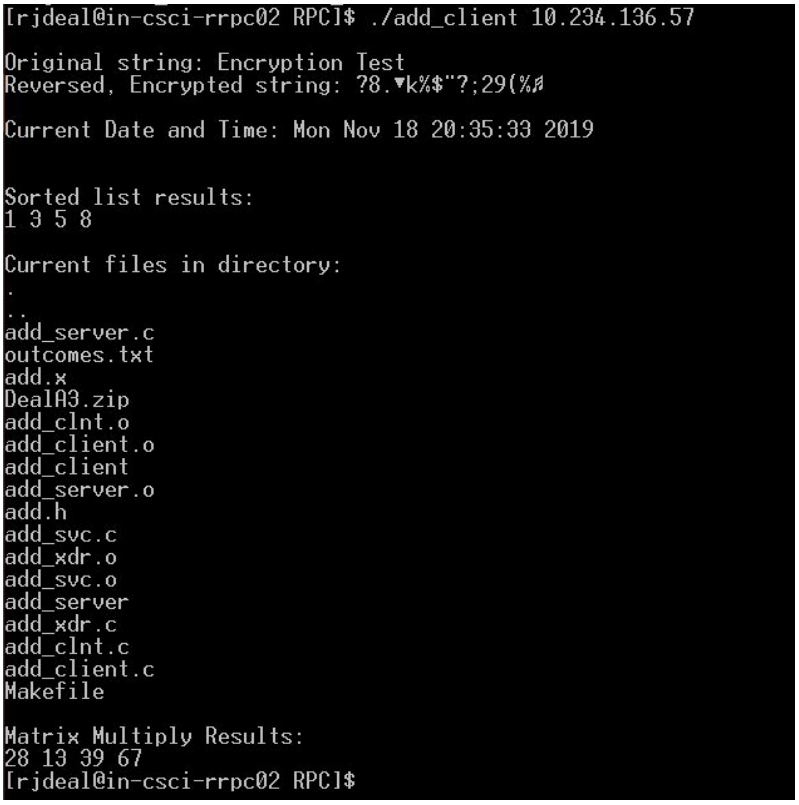
Compilation in C++

```
[rjdeal@in-csci-rrpc02 RPC]$ make
g++ -g -D_REENTRANT  -c -o add_clnt.o add_clnt.c
g++ -g -D_REENTRANT  -c -o add_client.o add_client.c
g++ -g -D_REENTRANT  -c -o add_xdr.o add_xdr.c
g++ -g -D_REENTRANT   -o add_client  add_clnt.o add_client.o add_xdr.o -lnsl -lpthread
g++ -g -D_REENTRANT  -c -o add_svc.o add_svc.c
g++ -g -D_REENTRANT  -c -o add_server.o add_server.c
g++ -g -D_REENTRANT   -o add_server  add_svc.o add_server.o add_xdr.o -lnsl -lpthread
[rjdeal@in-csci-rrpc02 RPC]$
```

Server-side execution



Client-side Execution



Final .x File

```
typedef int int_ptr<>;

struct Matrix {
  int_ptr matrix;
  int size1;
  int size2;
};

program ADD_PROG{
  version ADDS_VERS{
    string DateAndTime() = 1;
    int_ptr Sort(int_ptr) = 2;
    string List() = 3;
    Matrix MatrixMultiply(Matrix, Matrix) = 4;
    string ReverseEncryptedEcho(string) = 5;
  } = 1;
} = 0x30071315;
```

Function implementation

DateAndTime - DateAndTime is implemented via the C++ std::chrono library. I wanted to return a simple string in order to not have to deal with more XDR typing. Since I was unsure how to correctly convert the time point object returned by chrono into a string, I simply used a built in function to convert it into a timt_t type which can be converted into a string via the ctime function call.

```
bool_t
dateandtime_1_svc(char **result, struct svc_req *rqstp)
{
  std::cout << "DateAndTime called" << std::endl;
  bool_t retval = 1;
  //get the current clock time, convert it to ctime since its easy to convert
  //from ctime to string, return.
  auto time = std::chrono::system_clock::now();
  std::time_t t = std::chrono::system_clock::to_time_t(time);
  *result = (char *) malloc(256);
  strcpy(*result, std::ctime(&t));

  return retval;
}
```

Sort - Sort is implemented by doing a simple inplace bubble sort on the incoming arguments.This sorted list is then copied into the result.

```
bool_t
sort_1_svc(int_ptr arg1, int_ptr *result,  struct svc_req *rqstp)
{
  std::cout << "Sort called" << std::endl;
  bool_t retval = 1;
  // sort the list from arg1, copy into result
  int size = arg1.int_ptr_len;
  result->int_ptr_len = size;
  result->int_ptr_val = (int*)malloc(sizeof(int)*size);
  for(int i = 0; i < size; i++) {
    for(int j = 0; j < size; j++) {
      if (arg1.int_ptr_val[i] < arg1.int_ptr_val[j]) {
        int tmp = arg1.int_ptr_val[j];
        arg1.int_ptr_val[j] = arg1.int_ptr_val[i];
        arg1.int_ptr_val[i] = tmp;
      }
    }
  }
  for(int i = 0; i < size; i++) {
    result->int_ptr_val[i] = arg1.int_ptr_val[i];
  }
  return retval;
}
```

List - List uses the c-style directory access using dirent.h, unfortunately the easier C++ style wasn't added until C++17. This method simple opens the current directory, and reads a file from the directory until it reaches NULL (no files left). Each files name is added to a string and returned to the client.

```
bool_t
list_1_svc(char **result, struct svc_req *rqstp)
{
  std::cout << "List called" << std::endl;
  bool_t retval = 1;
  //use c style directory access as c++ doesn't add an easy one until c++17
  //which isnt available on server

  //while there is still a file to read in directory, add its name to string
  //copy the string into result
  std::string s = "";
  DIR* d = opendir(".");
  struct dirent * file;
  while((file = readdir(d)) != NULL) {
    s += file->d_name;
    s += "\n";
  }
  closedir(d);
  // note this is a magic number and WOULD break if enough files are in the
  // directory. I was having issues with using sizeof(s), etc...needs further
  // research, could be related to number of \n"
  *result = (char*) malloc(1000);

  strcpy(*result, s.c_str());
  return retval;
}
```

MatrixMultiply - This is a simple n^3 matrix multiply operation. Simple iterate over the original two matrices and store the results in the result matrix.

```
bool_t
matrixmultiply_1_svc(Matrix arg1, Matrix arg2, Matrix *result,  struct svc_req *rqstp)
{
  std::cout << "MatrixMultiply Called" << std::endl;
  bool_t retval = 1;
  // simple n^3 matrix multiply. Somewhat optimized as ikj loop is the best
  // for memory locality in a 1D array. Could be further optimized via registers, etc.
  result->size1= arg1.size1;
  result->size2= arg2.size2;
  result->matrix.int_ptr_len = result->size1*result->size2;
  result->matrix.int_ptr_val = (int*)malloc(result->matrix.int_ptr_len*sizeof(int));

  for (int loc = 0; loc < result->matrix.int_ptr_len; loc++) {
    result->matrix.int_ptr_val[loc] = 0;
  }

  for (int i = 0; i < arg1.size1; ++i) {
    for (int k = 0; k < arg2.size2; ++k) {
      for (int j = 0; j < arg1.size2; j++) {
        result->matrix.int_ptr_val[i * arg2.size2 + j] += arg1.matrix.int_ptr_val[i*arg2.size2+k] * arg2.matrix.int_ptr_val[k * arg2.size2 + j];
      }
    }
  }
  return retval;
}
```

ReverseEncryptedEcho - Stores the argument passed back into a string in order to utilize std::reverse easily. Next apply the encryption technique I used, which is the XOR cipher (https://en.wikipedia.org/wiki/XOR_cipher). I chose this technique because we have touched on it during my networking class, and it was simple and pretty interesting. The basic idea is you need some key, in my case ‘K’. Next, iterate over the original message and XOR the key with each character in the original message. This gets you the ciphered string. Interestingly enough, in order to undo the cipher you simple need to XOR the key with each character in the original message again. This means decryption and encryption are both very simple! Notably, using a very short key as I have done in this example can result in a less secure encryption, but that is outside the scope of the assignment.

```
bool_t
reverseencryptedecho_1_svc(char *arg1, char **result,  struct svc_req *rqstp)
{
  std::cout << "ReverseEncryptedEcho Called" << std::endl;
  bool_t retval = 1;
  // very simple XOR cipher https://en.wikipedia.org/wiki/XOR_cipher
  // essentially just do an XOR between the key with each character of the string.
  // Decryption is simple as you just apply the operation again with the same key.
  char key = 'K';
  std::string s = arg1;

  std::reverse(s.begin(), s.end());
  for (int i = 0; i < s.size(); i++) {
    s[i] ^= key;
  }
  *result = (char*) malloc(sizeof(s));
  strcpy(*result, s.c_str());
  return retval;
}
```

Conclusion

My conclusion is that learning some basics of RPC is good experience, but it would probably be better done with a more modern approach that is better supported by the tools we are given. Even if my analysis of the situation is wrong; and it could be, the amount of time spent getting to this point is significant with questionable results. Getting a non-threaded, non-concurrent implementation was fairly trivial within a few hours of reading, but getting a concurrent working example within the Linux environment took several days of work, and even still results can be non-deterministic.