Design Document: Basic RPC Server

Cooper Faber
Cruzid:cwfaber

December 11, 2020

1 Goals

This goal of this program is to implement a multi-threaded RPC server, that can handle arithmetic operations and variable storage, with recursive name lookup and persistent storage. The server must also be able to handle multiple threads accessing the variable storage without any synchronization errors. Additionally, file services such as writing and creating files, as well as dumping variable values to a file are provided. A client will send the server a request, and the server must translate the request, complete it, and return a translated code to the client with it's results.

2 Design

Once our server receives a connection, it will dispatch a thread to the client requesting. The server will then /recv() the first two bytes the client is sending, and move them to a function that will branch based on what function they correspond to. Recv() is called on a buffer with one 8bit spot - and is always called on one bit, usually in a loop. The one available slot in the buffer is then assigned to a separate variable. It will also recv() an additional four bytes for the identifier, and send it back if it is successful. From there, the branching function will call a separate, operation function. The server will recv() the rest of the required data for the operation, then attempt to complete the operation if the requested data (file access and/or variable storage) is available. If not, the thread will wait on a semaphore. Once the operation is completed, a response will be sent to the client, and the thread will be re-allocated for future use. Our server will be constantly listening for further instructions/connections, so it may handle multiple threads at any given time. Rinse and repeat until someone kills the server.

2.1 Network Handling

Upon initialization, the server will create the amount of threads specified from the runtime arguments, then wait for a conneciton to assign a thread to. When the server finds a client, it will assign a thread to the client, and continue waiting for more connections. When a client finishes it's process, the thread will be reallocated for another connection.

2.2 Linked List/Hash Table

The variable storage is represented via a chained hash table. The chained table is constructed from a linked list,and the DJBHash algorithm. Each node in the list contains a value, a name, and an index, which is given by the hash. The index is used for matching variables, as string functions in C add significant complexity and increase the resulting headache. Each key given to the hash will return an index, which we modulo by the number of buckets, which then corresponds to the bucket the key is sorted to.

2.3 Synchronization

This server uses two mutexes - one for the variable storage, and one for the threads. The thread mutex was provided, fully functional, in code from James Hughes - it functions to hold the threads in a wait() position until a connection is achieved, and the thread is assigned a cl number. The varstore mutex is posted on initiliazation, and closed whenever a thread needs to access the varstore. It is promptly re-opened when the thread finishes its section of process that involves the varstore.

2.4 Recursive Name Resolution

As suggested in the assignment, each node in our linked lists will hold a single, union-ed value: either a string, which represents that this variable's value is stored in another variable, or an int64. When prompted to retrieve a value, our server will recursively work through the stored values in the table until it reaches a constant, or hits the try limit (given upon initialization).

2.5 Persistent Storage

To have a persistent version of our variable storage, we will create a single file with similar syntax to a normal loaded file(i.e x=y). Every time a value is entered into the table, the server will print that x=y to the file. If a variable's value is edited, the file will instead just print the new x=y, since on launch, variables are read in order. If a variable x is deleted, the server will print x! which signifies variable deletion. As stated, upon launch, the server will iterate through the file, until it hits the end, adding each variable to the varstore.

2.6 Unit Functions

getbyte():when called, recv()s a single byte into the buffer that is passed
brancher(): given a 2-byte function code, will call respective math/fileop functions
putbytes(): given an array of bytes and a number to send, will send() bytes from array up to number
wirecode(): converts 8bit numbers to their respective 64 bit equivalent based on bit position
mathfunc(): takes function code, recv()s required bytes, does math operation, checks for overflow
read/write(): getbyte() required filename length, create array with length of filename, and retrieve the offset/buffer size. Then, if it is a write() op, it will getbyte() once, write that bit, and loop until all data is written. Read ops are similar - the server reads through a file to find it's length, then reads through again a byte at a time, sending each byte to the client.
load/dump(): as in r/w, get filename, then based on opcode, write/read from file. Key difference here is that the reads get loaded into varstorage, and writes also delete variables from varstore. **initLoad()**: runs when server starts - takes all values from file, and loads them into varstore

3 Main Pseudocode

```
Functions: wirecode(), readfunc(), writefunc(), mathfunc(), etc
Arguments: N = threads, H = buckets, Port = portnumber
 Initialization: Initialize connection at address X (hard coded to
 localhost) and port Y, arrays for function code,identifier,buffer,and
 return buffer
Create N Threads;
Initialize Chained Hash table with H buckets while forever do
    wait for connection;
    accept, assign to thread;
    process thread, as in assignment 1;
    when process finishes, clear thread;
end
```

3.1 functions

```
function BRANCHER(uint8 b1,b2,Hash H)
    getbyte() 6 bytes;
    assign function code/identifier to respective array vars
    putbytes(indentifier)
    Switch: send key,buf,hash to math() or respective fileop() depending
on b1;
    return value given from branch;
end function
function CREATE/SIZE(uint8$_t$ $* buf, uint8_t * return$)
    getbyte() twice to get length
    convert length to uint16
    getbyte() length times
    assign to local vars
    convert filename to string
    if create then
        create file with filename
else
        verify file exists;
        get uint64$_t size$
end function
function R/W(uint8$_t$ $* buf$)
    getbyte() twice to get length
    convert length to uint16
    getbyte() length times
    assign to local vars
    convert filename to string
    getbyte()x8
    convert offset to uint64
    getbyte()x2
    convert buffsize to uint16
    verify that file is not in use, and lock semaphore for readers/writers
if read then
        execute read twice - once to get file length, second time to send
    bytes to client
else
    getbyte buffsize times;
    write each byte as received
else
    getbyte once, write to file;
    repeat buffsize amount of times;
    unlock file semaphore
end function
```

```
function MATH(uint8_t $key, uint8_t * buf, uint8_t * return$)
    Switch: figure out which operation (sub/add/mult/div/mod),
variable conditions, and whether it is recursive;
    recv() additional bytes based on function and variable code
    if non-recursive call, given var with recursive resolution, fail process
    if recursive call, find value of each function using recursive lookup
    call wirecode() to convert variable to int64;
    do math operation
    check for overflow
    attempt to place in variable store, if needed
    persist variable
    call function to separate 64-bit into 8x8-bit, send to return
    place return var code in return
    send full response
    return 0 if success,other if error
end function
function DUMP/LOAD(uint8_t $key, uint8_t * buf, uint8_t * return, HashH$)
    getbyte() twice to get name length;
    convert length to uint16;
    getbyte() length times;
    assign to local var;
    switch on opcode;
    read table into var storage or write table to file;
    send success/error response;
end function
function DEL(uint8_t $key, HashH$)
    getbyte() once to get length
    getbyte() length times
    assign to local vars
    convert varname to string
    attempt to delete var with varname;
    send success/error response;
end function
```

4 References

I used:

James Hughes's multi thread skeleton:
`https://piazza.com/class/kex6x7ets2p35c?cid=291`
DJB Hash:
`https://www.programmingalgorithms.com/algorithm/djb-hash/c/`