Design Document: multi-threaded rpcserver

Perry David Ralston Jr.
CruzID: pdralsto

November 18, 2020

1 Goals

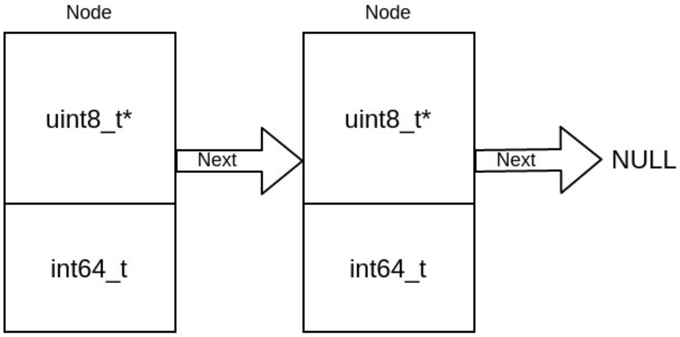
This multi-threaded RPC Server will build off of the design of the previous RPC server. The design for the origianl RPC Server can be found at the end of this document. The design from there remains unchanged unless specifically called out in this design document. The goal of this project is to create an RPC server capable of handling simultaneous requests from numerous clients, leveraging the synchronization techniques learned in lecture. Noted additions to this server is the dispatch thread, responsible for assigning worker threads to the inbound connections, and a shared key-value store for managing the store of variable values that all of the threads can access and modify in a semi-volatile manner. The key-value store persists across all connections to the server and is readable/editable by any thread with coordination.

2 Hash Table

For this project, I will be reviving and heavily modifying a hash table that I coded in March of 2017. The key structure of the hash table is a fixed size array of Node*(s) that make up a linked list. The nodes themselves contain a name stored as a char* and a value, int64_t. The hashing is done using a basic hashing function with a fixed decimal constant. The hash table supports the following public functions: insert, replace, delete, clear, dump, and load. The algorithims are detailed below.

2.1 Node Structure and Functions

Nodes are structured as a key value pair with a pointer to the next node in the list. Nodes are inserted into the hash table by hashing their key to acquire the index for the list that they will be inserted into. See diagram below.



Singly Linked List

//TODO figure out the synchronization process.

2.2 Hash Table Functions

```
genHash
Input uint8_t* key: string key to be hashed
Return int32_t: hashed_value corresponding this key's linked list location
    iterate string and add all char values together
    store sum in key
    return floor(tblSize * ((key * HASHCONST) - floor(key * HASHCONST)))
```

Hashing Function

```
Insert
Input uint8_t* key: key to insert into this hash table
Input int64_t value: value assigned to key
    hash = genHash(key)
    make new_node containing key and value
    if this[hash] is empty
        this[hash] = new_node
    else
        parse this[hash]
            if next is null
                next = new_node
                return
            if next.key == key
                temp = next
                next = new_node
                new_node.next = temp.next
                free(temp)
                return
```

Insert

```
replace
Input uint8_t* key: key to replace in this hash table
Input int64_t value: value assigned to key
    insert(key, size, value)
```

Replace

```
remove
Input uint8_t* key: key to delete from the table
    hash = genHash(key)
    if hash_table[hash] is empty
        return
    current = hash_table[hash]
    while current != null
        if current.key == key
            if follower == null
                hash_table[hash] = current.next
            else
                follower.next = current.next
            delete current
            return
```

Remove

```
lookup
Input uint8_t* key: key to find a value for
Input int64_t* value: Where to store the found value
Return int8_t: 0 if value was found successfully, -1 otherwise
    hash = gen_hash(key)
    parse this[hash]
        if current.key == key
            value = current.value
            return 0
    return -1
```

Lookup

```
clear
    parse hash_table
        current = hash_table[i]
        while current != null
            leader = current.next
            delete current
            current = leader
        hash_table[i] = null
```

Clear

```
dump
Input const char* filename: the file to save the contents of this hash_table to
Output int8_t: 0 if dump is successful, -1 otherwise, errno is set accordingly
    open(filename)
    //check for errors
    parse hash_table
        current = hash_table[i]
        while current != null
            //write key=value to filename
            //check for errors
            current = current.next
```

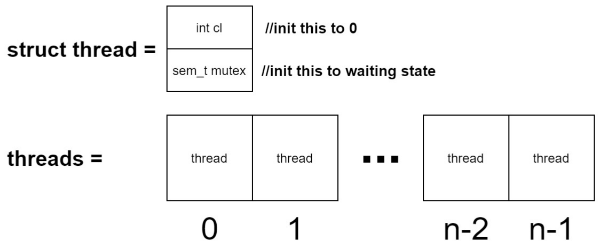
Dump

```
load
Input uint8_t* filename: the file to load the contents of this hash_table from
Output int8_t: 0 if load is successful, -1 otherwise, errno is set accordingly
    open(filename)
    //check for errors
    until eof is reached
        read file into buffer
        //check for errors
        while buffer has data
            extract whitespace seperated value from buffer
            seperate value on '=' char
            this.insert(lh_value, rh_value)
```

Load

3 Multi-Threading

In the previous rpcserver, request handling was limited to a single request at a time. Using the shared hash table described above and the pthread library, this rpcserver will have the ability to servce -N clients at the same time. -N is a command line argument that denotes the number of threads that the server should initially service. The default N value is 4. To achieve this, a structure will be created to house the thread references and act as a means of communication between the dispatch thread and the worker threads. The over all design is pictured below:



```
Main Thread {
        Create threads[n] where n = -N argv
        Create mainMutex, init to running state

        Repeat n times
            Init threads[i] with a new struct thread
            Call pthread_create, pass it start and threads[i]
        Initialize the network
        Infinite loop
            cl = accept(sock)
            While (i = findWorker() == -1) wait(mainMutex)
            threads[i].cl = cl
            signal(threads[i].mutex)

}
```

findWorker returns available thread or -1

```
start(void* arg) {

        self = (struct thread*) arg
        While (self -> cl == 0) wait(self->mutex)
        process(self->CL) //until eof
        self->cl = 0
        signal(mainMutex)
}
```

process is the original rpcserver code after init

This diagram was made using information
made available in CSE-130 discussion by James Hughes 11/16/2020

Design Document: rpcserver

Perry David Ralston Jr.
CruzID: pdralsto

October 30, 2020

1 Goals

This RPC Server will handle requests from a client to perform simple arithmetic: addition, subtraction, and multiplication (64 bit width) and basic file operations: read, write, create, and filesize query. All messages to the server are assumed to be in network byte order and all responses from the server will be formatted the same.

2 Initialization

Server initialization is handled at the command line by running rpcserver with the arg <host_name>:<port>. Using code retrieved from CSE130's Canvas Page. The command line arg is split at the ':' and the hostname and port are placed in the appropriate positions. The hostname must be $\leq$ 1 kB in size and the server will crash out otherwise. Once the socket has been created and bound, rpcserver runs an infinte loop listening on the address and port specified at the command line.

2.1 Handling Requests and Sending Responses

Requests from the client are assumed to be in network byte order. Responses will be formatted in network byte order prior to sending responses "over-the-wire". The subroutines shown below detail the process for converting to and from network byte order. This server makes no assumptions about the rate at which all of the data is received from the client. Messages from the client are read into a bounded_buffer object for processing.

```
Input Bounded_Buffer& bound_buff:  reference to a Bounded_Buffer object
Input var_int_type to_convert: variable width integer type, specified at time of call
Input int cl: client file descriptor needed for pushByte call

for i in (sizeof(to_convert) ... 0]
    to_push = (to_convert >> 8i) & 0xFF
    bound_buff.pushByte(cl, to_push)
```

Convert to Network Byte Order

```
Input Bounded_Buffer&  nbo_array: Bytes to be converted
Input int cl: client file descriptor needed for getByte call
Output var_int_type: variable width integer type

for i in [0 ... sizeOf(var_int_type))
    converted = (converted << 8) + b_buff.getByte(cl)
return converted
```

Convert from Network Byte Order

2.2 Bounded_Buffer Class

The Bounded_Buffer class is responsible for maintaining, filling, and flushing the bounded buffer used to store messages to and from the client for internal processing. It has private members for three uint8_t pointers that maintain the root, start and end of the buffer and public functions empty, fill, flush, getByte, getBytes, pushByte, and pushBytes. The public functions are detailed below:

```
empty()
Output boolean: true if there is no data to read from the
                buffer
    return start == end

fill()
Input cl: client file descriptor
Output: returns bytes read
    if (start != root)
        start = root
    recv up to BUFFER_SIZE number of bytes into buffer from cl
    if bytes_read == -1
        log the error and exit
    end = start + bytes_read + 1

flush()
Input cl: client file descriptor
```

```
Output: returns bytes sent
    if start == end
        return 0
    write end - start bytes to cl
    start = end = root
    if bytes_sent = -1
        log the error and exit
    return bytes_sent

getByte()
Input cl: client file descriptor
Output: return pointer to the next byte available in the array
        or NULL if no bytes are available after attempting
        to fill the buffer from cl
    if empty
        fill
    if empty
        return NULL
    ret_value = start[0]
    ++start
    return ret_value

getBytes()
Input cl: client file descriptor
Input size_t size: number of bytes to read from the buffer
Input uint8_t* dest: destination array
Output int8_t: 0 if all requested bytes could be read -1 otherwise
    for i in [0 ... size)
        currByte = getByte()
        if currByte == NULL
            return -1
    dest[i] = getByte()
    return 0

pushByte()
Input cl: client file descriptor
Input uint8_t in_byte: the byte to be placed in the buffer
Output int8_t: 0 if byte was written successfully, -1 otherwise
    if end > root + BUFFER_SIZE
        flush(cl)
    end[0] = in_byte
    ++end

pushBytes()
Input cl: client file descriptor
Input uint8_t* in_bytes: the bytes to be placed in the buffer
Input size_t size: the number of bytes to be placed in the BufferError
```

```
Output int8_t: 0 if size bytes was written successfully, -1 otherwise
    if size < remaining capacity
        flush
        if not empty
            return -1
    for i in [0 ... size)
        if pushByte(cl, in_bytes[i]) == -1
            return -1
    return 0
```

Bounded_Buffer Public Functions

2.3 Resolving Arguments and Calling Functions

Once the request has been parsed, the corresponding function call is made. If no corresponding function call can be found then response header containing EBADRQC is sent back to the client. Arguments to the corresponding function are parsed from the data portion of the request.

2.4 Supported Functions

Math Functions Add, Subtract, and Multiply are supported

add Add two numbers, A and B, together returning the value. If overflow would occur set err_code to EINVAL(22)

```
add
Input int64_t a: number to add to b
Input int64_t b: number to add to a
output int64_t: result of a + b
    set errno to 0
    if result will overflow
        set errno to EINVAL
        return EINVAL
    result = a + b
    return result
```

subtract Subtract B from A, returning the value. If overflow would occur set err_code to EINVAL(22)

```
subtract
Input int64_t a: number to subtract b from
Input int64_t b: number to subtract from a
output int64_t: result of a - b
    set errno to 0
    if result will overflow
        set errno to EINVAL
        return EINVAL
    result = a - b
    return result
```

multiply Add two numbers, A and B, together returning the value. If overflow would occur set err_code to EINVAL(22)

```
multiply
Input int64_t a: number to multiply by b
Input int64_t b: number to multiply by a
output int64_t: result of a * b
    set errno to 0
    if result will overflow
        set errno to EINVAL
        return EINVAL
    result = a * b
    return result
```

File Functions Read, Write, Create, and Filesize are supported

read Read bufsize bytes from file into buffer starting at the offset and return the number of bytes read if there was no error, -1 otherwise.

```
read_file
Input char* filename: file to return the size of
Input uint64_t offset: where to start reading from
Input uint16_t bufsize: how many bytes to read from the file

    file_size = filesize(filename)
    if file_size == -1
        set header error status
        send header
        return -1
    if filesize - offset < bufsize
        set header status
        send header
        return -1
    file_d = open(filename, O_RDONLY)
    if  file_d == -1
        set header error status
        send header
        return -1
    send header
    while bytes_read < bufsize
        curr_read = read(file_d, buffer, BUFFER_SIZE)
        if curr_read == -1 or == 0
            close connection
            return -1
        if bytes_read + curr_read <= bufsize
            send curr_read bytes to client
            bytes_read += curr_read
        else
            send bufsize - bytes_read to client
            return bufsize
    return 0
```

write Write buffsize bytes from buffer to a file starting at offset and return the number of bytes written if there was no error, -1 otherwise.

```
write_file
Input char* filename: file to return the size of
Input uint64_t offset: where to start reading from
Input uint16_t bufsize: how many bytes to write to the file
    file_d = open(filename, O_WRONLY)
    if  file_d == -1
        set header error status
        send header
        return -1
    send header
    while bytes_written < bufsize
        fill buffer up to bufsize - bytes_written
        if no bytes written to buffer
            close connection
            return -1
        write filled_bytes to filename
        bytes_written += filled_bytes
    return 0
```

create Create a new 0 byte file if it does not already exist, returns -1 if an error occurs

```
create_file
Input char* filename: file to create
Output int64_t: 0 if successful, -1 otherwise
    if open(filename, O_CREATE, O_EXCL, O_WRONLY) == -1
        set header error status
        send header
        return -1
    send header
    return 0
```

filesize Returns the size of an existing file, -1 in the event of an error

```
get_filesize
Input char* filename: file to return the size of
    if stat(filename, fileStats) == -1
        set header error status
        send header
        return
    return fileStats.st_size
```