Design Document
Persistant Multi-Threaded RPCServer

Perry David Ralston Jr.
CruzID: pdralsto

December 11, 2020

1 Goals

1. Fix the bugs from asignment 2
2. Support synchronized storing of values or variable names in key/value store
3. Support recursive name resolution to a specified max depth
4. Support large scalability
5. Key-Value store will be persistant across instantiations of the server

2 Syncronized Hashtable

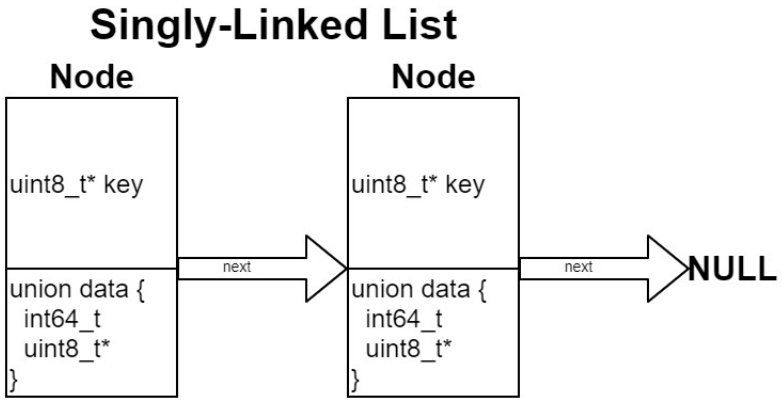
The hashtable defined below utilizes the djb2 hashing function, sourced from http://www.cse.yorku.ca/~oz/hash.html on 11/23/2020. It supports these public functions:

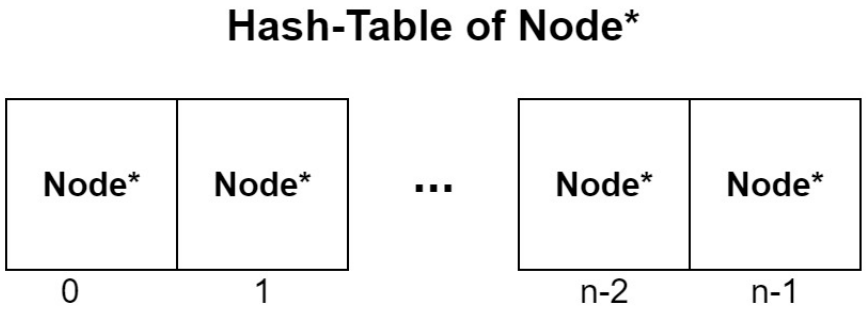
insert	remove	lookup
clear	dump	load
acquire	release	

Data is stored as a Singly-Linked List of Node objects, detailed below. The hash table is wrapped by an inherited class, SyncHash, that adds index level synchronization to the table. The syncronization is managed using a parallel array of mutexes and all of the functions use calls to acquire and release to change the states of them.

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. Nodes are instantiated with their next pointers set to null. The data of the node is a union type int64_t and char*. This allows for the nodes to store either a variable name or a numeric value, supporting recursive name resolution. See diagram below.





2.2 Hash Table Functions

```
genHash
Input uint8_t* key: string key to be hashed
//sourced from http://www.cse.yorku.ca/~oz/hash.html 11/23/2020
Return int32_t: hashed_value corresponding this keys linked list location or -1 if the key is invalid
    if validate_key(key) == -1
        return -1
    key_value = 5381
    traverse key
        key_value = key_value * 33 + key[i]
    return floor(tblSize * ((key * HASHCONST) - floor(key * HASHCONST)))
```

Hashing Function

Acquire and release supports two different public interfaces, single and list. The list version acquires locks in index order to avoid dead lock.

```
acquire
Input uint8_t* key: key to acquire lock on
Input int64_t ident: unique thread identifier
Returns int8_t: 0 if successful. -1 If the key is invalid
  hash = genHash(key)
  if hash == -1
    return -1
  acquire(hash, ident)
  return 0
```

Acquire

```
acquire
Input uint8_t** keys: keys to acquire locks on
Input int64_t ident: unique thread identifier
Returns int8_t: 0 if successful. -1 If a key is invalid
  hashlist = {genHash(key) for key in keys}
  sort(hashlist)
  if hashlist[0] == -1
    return -1
  for hash in hashlist
    acquire(hash, ident)
  return 0
```

Acquire (list)

```
release
Input uint8_t* key: key to release lock on
Input int64_t ident: unique thread identifier
Returns int8_t: 0 if successful. -1 If the key is invalid
  hash = genHash(key)
  if hash == -1
    return -1
  if this thread owns the lock
    release(hash)
  return 0
```

Release

```
release
Input uint8_t** keylist: keys to release lock on
Input int64_t ident: unique thread identifier
Returns int8_t: 0 if successful. -1 If the key is invalid
```

```
  hashlist = {genHash(key) for key in keys}
  if hash == -1
    return -1
  if this thread owns the lock
    release(hash)
  return 0

  hashlist = {genHash(key) for key in keys}
  sort(hashlist)
  if hashlist[0] == -1
    return -1
  for hash in hashlist
    acquire(hash, ident)
  return 0
```

Release (List)

```
Insert
Input uint8_t* key: key to insert into this hash table
Input int64_t value: value assigned to key
Input int64_t ident: thread specific identifier
Return int8_t: 0 if insert is successful, -1 otherwise, errno
    is set appropriately
    hash = acquire(key, ident);
    if hash == -1
        return -1
    current = hashtable[hash]
    prev = hashtable[hash]
    while current != null
        if current.key == key
            current.data = value
            release(hash)
            return 0
        prev = current
        current = current.next
    create a newNode using key and value
    if prev == current
        hashtable[hash] = newNode
    else
        prev.next = newNode
    release(hash)
    return 0
```

Insert

```
remove
Input uint8_t* key: key to delete from the table
Input int64_t ident: thread specific identifier
```

```
Return int8_t: 0 if remove is successful, -1 otherwise, errno
    is set appropriately
    hash = acquire(key, ident)
    if hash == -1
        return -1
    current = hashtable[hash]
    if current is not null and current.key == key
        hashtable[hash] = current.next
        delete current
        return 0
    follower = current
    while current is not null
        if current.key == key
            follower.next = current.next
            delete current
            release(hash);
            return 0
        follower = current
        current = current.next
    release(hash)
    return -1
```

Remove

```
lookup
Input uint8_t* key: key to find a value for
Input int64_t* value: Where to store the found value
Input int64_t ident: thread specific identifier
Return int8_t: 0 if value was found successfully, -1 otherwise
    hash = acquire(key, ident)
    traverse hashtable[hash]
        if current.key == key
            value = current.value
            signal(hash)
            return 0
    release(hash)
    return -1
```

Lookup

```
clear
    acquire all of the locks
    traverse hashtable
        if hashtable[i] is defined
            current = hashtable[i]
            while current is defined
                temp = current
                current = current.next
```

```
                delete temp
            hashtable[i] = null
        release(i)
```

Clear

```
dump
Input char* filename: file to print the key-value store to
  acquire all of the locks
    traverse hashtable
        if hashtable[i] is defined
            current = hashtable[i]
            while current is defined
                print key=value\n
        release(i)
```

Dump

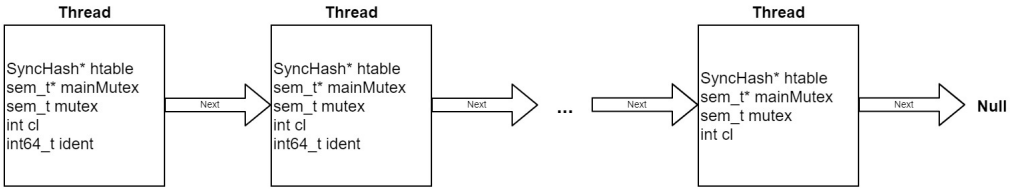
```
load
Input uint8_t* filename: the file to load the contents of this
    hash_table from
Input int64_t ident: uniqure thread identifier
Output int8_t: 0 if load is successful, -1 otherwise, errno is
    set accordingly
    open the file named filename
    //error handling
    acquire all locks
    until eof is reached
        read a line from the file
        //error handling
        parse the line into key and value
        //error handling
        insert(key, value, ident)
    release all locks
    return 0
```

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:



Threads

The instructions for request handling are moved into a new function, process, that recieves a Thread& as an argument. The Thread type is shown above and the members are used by the process function in order to facilitate synchronization of the access to the hashtable as well as retrieving and responding to the request from the client.

```
process
Input Thread* self:  contains all of the information associated
    with this Thread
Reliant data:
  math_funcs:          Array of math function pointers to reduce
    repeated code
  resp_header:         struct representing the response header
  buffer:              4kB bounded buffer for client/server data
    handling
  var_args:            3 index array of uint8_t* to hold keys of
    variable args from client
  ARG_COUNT:           Constant set to 3 for batch lock
   acquisition of args from the hashtable
  MATH_OPS:            0x0100
  FINAL_BYTE:          0X000f
  MATH_FUNC_COUNT:     5
  CLEAR_CONFIRM:       0x0badbad0
  DEL:                 0x010f
  GETV:                0x0108
  GETV:                0x0109
  DUMP:                0x0301
  LOAD:                0x0302
Notes: For brevity, error checking is assumed
  while client connection is open
    build_header(resp_header, buffer)
```

```
21      math_op = (header->op & MATH_OPS) + (header->op &
    FINAL_BYTE)
22      if math_op != 0
23        fun_index = (math_op & FINAL_BYTE) - 1
24        if fun_index < MATH_FUNC_COUNT
25          Determine the number and which args are variables,
    assign them to corresponding indicies of var_args
26          if var_args[2] != null
27            self->htable->acquire(var_args, ARG_COUNT, self->
    ident)
28          if var_arg[0] != null
29            self->htable->lookup(var_arg[0], arg1, self->ident)
30          if var_arg[1] != null
31            self->htable->lookup(var_arg[1], arg2, self->ident)
32          math_funcs[fun_index](arg1, arg2, result)
33          if var_args[2] != null
34            self->htable->insert(var_args[2], result, self->ident
    )
35          send response
36        else
37          if math_op == DEL
38            key = read key from buffer
39            self->htable->remove(key, ident)
40          else if math_op == GETV or SETV
41            key = read key from buffer
42            If recursive flag is set
43              resolved_key = recursive lookup result
44            else
45              resolved_key = single stage lookup result
46            if math_op == GETV
47              send response
48            else
49              key = read key from buffer
50              if key is valid
51                self->htable->insert(resolved_key, key, self->
    ident)
52                send response
53              else
54                resp_header.error = EINVAL
55                send response
56          else
57            resp_header.error = EINVAL
58            send response
59      else
60        if header.op == one of the file operations
61          behavior is unchanged from asgn1 and 2
62        else if header.op == DUMP
63          filename = filename retrieved from buffer
64          self->htable->dump(filename, self->ident)
```

```
65          send response
66        else if header.op == LOAD
67          filename = filename retrieved from buffer
68          self->htable->load(filename, self->ident)
69          send response
70        else if header.op == CLEAR
71          arg1 = pull 32 bit unsigned int from buffer
72          if arg1 == CLEAR_CONFIRM
73            self->htable->clear(self->ident)
74            send response
75          else
76            resp_header.error = EINVAL
77            send response
78        else
79          resp_header.error = EINVAL
80          send response
```

Process

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
```

```
65 Output int8_t: 0 if size bytes was written successfully, -1
       otherwise
66     if size < remaining capacity
67         flush
68         if not empty
69             return -1
70     for i in [0 ... size)
71         if pushByte(cl, in_bytes[i]) == -1
72             return -1
73     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
```