CS454 A3 - Load balancer

Anton Pravdin - 20378263 - apravdin

Serghei Filippov - 20390471 - sfilippo

Usage

NOTE: The information on this page can also be found in the README file.

Requirements

pthread - both for the RPC lib and binder executable

Compiling

NOTE: The makefile will compile any .cc files found in the root directory
and link with all the .o files found in the src directory
and link with librpc
The respective executable will have the extension .out

> cd ROOT_DIR
> make

This will generate `./binder.out`, `./server.out`, `./client1.out` and `src/librpc`

Cleaning

> cd ROOT_DIR
> make clean

Running

> ./binder.out
This will start the binder and display the `BINDER_ADDRESS` and `BINDER_PORT`.

> ./server.out
This will start up the example server provided with this assignment.

> ./client1.out
Same as server except the client.

Protocol

The suggested protocol was implemented with a slight modification:

Length, Type, [MessageLen, Message]

Based on the type of the message the number of message segments following varies. Each of these message segments first specifies the length in the form of an integer and is followed by the respective number of bytes.

Although the Length section never ended up being used, it was planned to be used as a sanity check for the variable number of message segments.

Binder

The binder has a tcp acceptor very similar to that of rpcExecute. On a valid `accept` command the acceptor is to create a thread using the pthread library and pass along the socket descriptor of the connection.

The thread begins execution with the request handler function. Based on the type of the message INIT, LOOKUP, or TERMINATE, a respective function is called to handle the call. The reason that REGISTER is not part of the above list is because it is expected to arrive following an INIT request. Therefore calling REGISTER without calling INIT first will be ignored by the binder.

There are static objects used by the handlers: `registered_functions` and `server_ports`. The list container `server_ports` stored pairs of <server_addr, port>. This object contains all the active servers for the TERMINATE request. The `registered_functions` is a dictionary mapping function hashes to list of <server, port> pairs.

The `registered_functions` allows for a simple form of round robin by popping the first valid server reference from the list and then pushing it back at the bottom. Function overloading is handled via function hashes which are unique since they use the argument types when being generated.

The hashes are generated in `rpcCall` and `rpcRegister` before being sent to the binder using a shared function to ensure they match. The hash is a simple concatenation of the function name and the char array representation of the respective argument types. Note that the argument types get masked out to ignore array lengths and I/O specifications.

Due to the use of threads and shared resources, mutex locks were implemented to ensure data integrity. These locks were positioned as close as possible to the respective shared resources so as to reduce how long binder threads are blocked on each other.

The binder is also capable of determining the address of the peer that is connected to it via the socket descriptor.

The TERMINATE task is handled by sending a TERMINATE request to all server and then the binder process is terminated using the exit system call. Since the process exits all the threads created within it are guaranteed to terminate.

RPC Library

rpcCall

The following actions done for successful `rpcCall` commands.

1. connect to binder
2. generate function hash
3. LOOKUP request is sent to the binder
4. connect to server
5. arguments are parsed and copied into a continuous array
6. EXECUTE request is sent to the server
7. get server response
8. ARG_OUTPUT results are copied back into args

rpcInit

The following actions done for successful `rpcInit` commands.

1. connect to the binder
2. create a server connection acceptor
3. INIT request is sent to the binder

rpcRegister

The following actions done for successful `rpcRegister` commands.

1. make sure a connection to the binder exists
2. generate a function hash
3. register the function skeleton locally
4. REGISTER request is sent to the binder

rpcExecute

The following actions done for successful `rpcExecute` commands.

1. handler thread is created
2. if TERMINATE request received then process exits
3. if EXECUTE request received continue
4. receive name, argument types, and argument data

5. generates function hash
6. check against local registered skeletons
7. map pointers to argument data
8. run the skeleton
9. copy function call results into argument data array
10. EXECUTE_SUCCESS or EXECUTE_FAILURE is sent with the results
11. clean up thread

rpcTerminate

The following actions done for successful `rpcTerminate` commands.

1. connect to binder
2. TERMINATE request is sent

rpcCacheCall

Not implemented

Error Codes

The error codes are defined in include/rpc_errno.h

Errno Name	Code	Description
RETVAL_SUCCESS	0	the default “all good” return value
ERRNO_ENV_VAR_NOT_SET	-1	missing BINDER environment variables
ERRNO_FAILED_TO_CONNECT	-2	socket fails to connect to ADDR:PORT
ERRNO_FAILED_READ	-3	a read fails
ERRNO_FUNC_FAILED_SEND	-4	a send fails or sends the wrong number of bytes
ERRNO_NO_SPACE	-5	failed to allocate memory
ERRNO_FUNC_NOT_FOUND	-10	LOOKUP failed
ERRNO_INIT_FAILED	-11	INIT failed
ERRNO_FAILED_TO_START_SERVER	-12	server connection acceptor failed to start
ERRNO_REGISTER_FAILED	-13	REGISTER failed
ERRNO_EXECUTE_FAILED	-14	EXECUTE failed
BINDER_INVALID_COMMAND	-20	binder received an invalid request

Places for Improvements

There were unnecessary memcpy operations during argument parsing. Due to the current design these were needed, but if the `rcpCall` did not send all the data as one continuous array then it could have been handled through many separate send commands. Similarly the result could be read directly into the address the client specified.

The binder implements a slightly different form of round-robin. It currently performs round-robin on individual function calls. There it is possible for the load to not be distributed evenly. In order the solve this issue, one possible solution is to store the server references in a circular buffer and offer each server a change to execute in turn.