CS454 Assignment #3 Documentation

Binder

Data Sturctures

Server contains socket , port, and id of the server, which is the information the server provides to the binder. We chose the container for registered servers to be a std::set. Servers that are registered are added to the set. Set is a good choice for us, since the order that the servers have been added do not matter.

FunctionSignatureAndServer is a contains the function signature and the server the corresponding function is located on. We kept track of functions and the server they refer to by having a vector of FunctionSignatureAndServer, since vectors are easy to use and we can iterate through them

Implementation

There is an iterator for the past called function on any server. The iterator is moved to the function when it gets called, such that the next time an RPC call to a function happens, the iterator starts from that function and attempts to find the new function. This way a simple **round-robin** is implemented for RPC call to functions on servers.

If the binder receives a function with the same name and arguments (ignoring argument length) as another function on the same server, the binder ignores the new function and a warning code is returned in the response message. Otherwise, the binder registers the new method, even if the name of the new method is the name of another method on the server it is located. This is how **function-overloading** is implemented.

When the binder receives a **TERMINATE** message, it sends a message to all servers with a TERMINATE type and it turns it's terminate flag from false to true. The binder remains active until all servers terminate (which the binder detects through the select loop). When all servers terminate, the binder terminates.

Client

Data Structures

server is a typedef for pair<string, string>. which is the hostname and port.

The **client cache** is a std::map<server, set<FunctionSignature> >. That is, it maps a server to a set of function signatures (see server section for FunctionSignature def'n) This is the most logical respresentation for this data, as it prevents duplicate servers or signatures from being registered for 'free'.

RpcCall

Connects to the binder, sends a LOQ_REQUEST message to get a response with a server location. Sends an EXECUTE message to the server with the name, argTypes and args. The args must each be dereferenced before they can be sent.

RpcCacheCall

Attempt to connect to a server in the cache. On each failure, remove that server from the cache. If no connection can be made this way, send a LOC_CACHE_REQUEST to the binder and add the returned server locations to the cache. If none were returned, we fail with an error, otherwise connect to one of the servers returned. From this point, execution is the same as rpcCall.

RpcTerminate

Just Sends a TERMINATE message to the binder then returns

Server

Data Structures

ArgType is constructed using an integer argType and parses it (by bit-shifting) into its four parts: input, output, type and arrayLength. It uses these parts to implement relational operators == and < to accomodate function overloading. In particular, the arrayLength is only compared as scalar vs. non-scalar (instead of using the absolute length)

FunctionSignature is composed of an ArgType and a name and implements relational operators == and < to accomodate function overloading.

The **server database** is just a simple std::map[FunctionSignature, skeleton]. FunctionSignature stores the function name and an ArgType (a custom struct used to parse an integer arg type). FunctionSignature and ArgType each implement operator< so they can be used as the key to the map. Using this data structure ensures uniqueness in the functions registered automatically.

Methods

RpcInit

Creates a listening socket for clients and then opens a connection to the binder.

RpcRegister

Sends a REGISTER message to the binder and adds the FunctionSignature to the local database. The binder also sends a return value in its response.

RpcExecute

Starts up the main accept loop, waiting for connections from clients. When data arrives from the client, the message type is checked and an appropriate handler is called.

For an EXECUTE message, a new thread is spawned and the accept loop continues right away. The new thread allocates memory for the args in the message, assigns the inputs passed from the client to the allocated memory, finds a skeleton in the local database, executes the skeleton with the arguments passed in the message, then sends a response to the client with the output values. The allocated memory is then freed. Note that a copy of original pointers to allocated memory is kept for this delete in case the skeleton modified any of the args pointers.

For a TERMINATE message, the server checks if the socket the message was received on is the same one opened to the binder in rpcInit. Then it sets a flag variable to exit the main accept loop. When the main loop exits, it waits for all executing threads to finish and closes all open sockets before returning.

The threads are spawned as detached threads and a shared counter keeps track of the number of threads, so that the server can wait for this counter to reach zero before exiting. The counter is incremented for each thread created and each thread decrements the counter when it finishes executing.

Marshalling/Unmarshalling

Short/Int/Float/Double: These are all converted to a string representation then sent over the write as a null terminated string. This has the advantage of not worrying about endianess, or any internal represention differences specific to any machine.

Char: Just sent as a single char

Args: When arguments are sent client->server, only the input arguments are sent. When arguments are sent client->server, only the output arguments are sent. This saves bandwidth of sending unecessary information. The associated argTypes are also sent first

accross the wire because the array lengths are needed on the other end to deserialize the args

Error Codes

Error codes are defined as enum values in error_code.h:

The following are returned when an error is returned from a call to socket(), bind(), listen(), send() or recv() or select():

SYS_SOCKET_ERROR = -98

SYS_BIND_ERROR = -97

SYS_LISTEN_ERROR = -96

SYS_SEND_ERROR = -95

SYS_RECV_ERROR = -94

SYS_SELECT_ERROR = -93

MISSING_ENV_VAR = -94 is returned when the binder environment variables that are required are not set

INVALID_ADDRESS = -93 is returned when trying to open a connection fails because the connection is refused

WRONG_MESSAGE_TYPE = -92 is returned if an invalid message type is received by a client, server or binder. This will not happen under normal usage.

These two are returned if something goes wrong while sending or recieving an entire message:

MSG_SEND_ERROR = -93

MSG_RECV_ERROR = -92

// Server

NOT_INITIALIZED = -91 is returned if calling rpcRegister() or rpcExecute() before rpcInit()

NO_REGISTERED_METHODS = -90 is returned from rpcExecute()if it is called before at least one successful call to rpcRegister()

BINDER_DIED = -89 is returned from rpcExecute() if the connection to the binder is lost before a TERMINATE message is received

NOT_REGISTERED_ON_SERVER = -88 is returned from rpcCall() if the server cannot find the requested method in its local database

SKELETON_EXCEPTION = -87 is returned from rpcCall if the skeleton threw an exception during its execution

SKELETON_ERROR = -86 is returned from rpcCall() if the skeleton returned a error (<0) on the server (this masks the int return value of the skeleton, since it is not supposed to return a value)

// Binder

NOT_REGISTERED_ON_BINDER = -85 is returned from rpcCall() if the binder cannot find a method in its database

// Warnings

SKELETON_WARNING = 1 is returned from rpcCall() if the skeleton returned a warning (>0) on the server (this masks the int return value of the skeleton, since it is not supposed to return a value)

ALREADY_REGISTERED = 1 is returned from rpcRegister() is a server tries to register the same method twice