CS 454 A3 System Manual

System Design

1. Function Overloading

   To accomplish function overloading, each function is hashed to a string using its name and argument types. Two functions with the same name but different signature will be hashed to two different strings. Then when checking to see if there exists a function with the desired name and parameters, we simply just look for the hashed string.

2. Binder Database & Round Robin Scheduling

   The database is a C++ vector of structs. Each structure contains a server's host name and port, and a set of function signature strings corresponding to the functions registered by that particular server. Then to check whether a duplicate entry exists for a server, we just look for the entry with the server's name and port and check to see whether the function signature exists in the set. When it comes to removing the entries for a particular server, we just have to remove the one vector element corresponding to the server.

   In order to implement round robin scheduling, we keep a global index into the database vector. When the client makes a location request to the binder, we check to see whether the entry at the current index contains the hashed signature for the function in the location request. If so, we just return the location info stored in that entry. Otherwise, we increment the index (modulo the size of the vector) and check again. If each entry has been checked, then there is no server available to satisfy the request so we return the appropriate failure message.

3. Messaging & Marshalling

   We built a custom message class to do all the sending, receiving and marshalling. In order to send data, we just call the setters to set the type of the message and create deep copies of all the data. Then we call the send function which will automatically marshal the parameters and send the data to a specified socket.

   In order to receive data, the user calls a function to receive the header followed by the message body. There are two ways of receiving data – blocking and non-blocking. A blocking call causes the current thread to block until the entirety of the message has been received. The non-blocking call reads all the data available at the current socket without causing the current thread to block; non-blocking receive may need to be invoked multiple times before the entire message is received and parsed. When receiving a message, the data is automatically unmarshalled from the message's format into the format specified by the client/server programs.

The message class uses C++ object-oriented concepts to do automatic memory management. The user does not have to deallocate any memory allocated by the class so no memory leaks can occur. In order to deal with errors, the class will throw an exception instead of returning an error code. This reduces the need for internal status codes and makes the resulting code cleaner – instead of having an if statement after every message send/receive call, the statements can be wrapped in a try-catch clause.

The message parameters are marshalled as follows:

- Function Name – sent as a 64-char string (including NULL terminator)
- Server Identifier – sent as a 48-char string (including NULL terminator)
- Port – sent as a 32-bit signed integer
- Reason Code – sent as a 32-bit signed integer
- Argument Types – sent as a 32-bit signed integer representing the number of arguments followed by a 32-bit signed integer for each argument type (however, the NULL terminator for the argument types is not sent since the number of arguments is known)
- Arguments – each argument is sent as an array of the corresponding type (non-array arguments are sent as an array of length 1)

This scheme allows us to always know the size of each argument before we read it from the socket (for arg types and args, we can read in the number of args/arg types and use it to determine the total size of the arg types/args).

Error Codes

```
WARNING_DUPLICATE_FUNCTION = 1,          // The server is attempting to register a function that has already been registered

ERROR_MISSING_ENV = -1,                  // The BINDER_ADDRESS or BINDER_PORT env variables have not been manually set
ERROR_ADDRINFO = -2,                     // If getaddrinfo fails, ie the getaddrinfo method returns a negative error code
ERROR_SOCKET_CREATE = -3,                // If creating a socket fails, ie the socket method returns a negative error code
ERROR_SOCKET_CONNECT = -4,               // If connecting to a socket fails, ie the connect method returns a negative error code
ERROR_SOCKET_BIND = -5,                  // If binding to a socket fails, ie the bind method returns a negative error code
ERROR_SOCKET_LISTEN = -6,                // If listening to a socket fails, ie the listen method returns a negative error code
ERROR_SOCKET_SELECT = -7,                // If select fails, ie the select method returns a negative error code
ERROR_SOCKET_ACCEPT = -8,                // If accept fails, ie the accept method returns a negative error code
ERROR_SOCKET_NAME = -9,                  // If getting the socket name fails, ie the getsockname method returns a negative error code
ERROR_HOSTNAME = -10,                    // If getting the host name fails, ie the gethostname method returns a negative error code
ERROR_MESSAGE_SEND = -11,                // If sending a message fails, ie the send method returns a negative error code
ERROR_MESSAGE_RECV = -12,                // If recieving a message fails, ie the recv method returns a negative error code
ERROR_MISSING_FUNCTION = -13,            // If the desired function does not exist on the server or binder
ERROR_FUNCTION_CALL = -14,               // If the function called returns a negative error code
ERROR_NOT_CONNECTED_BINDER = -15,        // The server is not connected to the binder, ie the binder socket has not been created on the server
ERROR_SERVER_NOT_RUNNING = -16,          // The server is not running, ie the socket for clients to connect to has not been created
ERROR_LOST_CONNECTION_BINDER = -17,      // The binder disconnected from the server
```

Unimplemented Functionality

None

Advanced Functionality

We implemented rpcCacheCall as a bonus feature. One way to test this would be for a client to make a successful cache call and then make a second call. Before the second call is made, the binder should be closed (not using TERMINATE because the servers should remain up). If the second cache call then succeeds, the caching works properly.