CS654 Assignment 3 Documentation

Group members: Andrew Palmer, Junyu Lai

Marshalling/unmarshalling of data

We implement a **Message** class, which handles the marshalling, and unmarshalling of messages.

Serialization and Deserialization

Serialization is implemented using fixed-length fields in a byte array for the whole message. Each index position has a specific meaning and the array has a specific structure that is shared across the system. The data will be serialized/deserialized to/from a sequence of characters, in the following order:

Message type (4 bytes) — Reason Code (4 bytes) — Port (4 bytes) — Host (128 bytes) — Name (64 bytes) — Arg Type (if applicable) — Arguments (if applicable)

The serialization/deserialization of arguments can be determined by the argument types.

The argument type sizes are specified as follows:

Argument type	Size (bytes)
Character	1
Short	2
Integer	4
Long	8
Double	8
Float	4

Structure of binder database

Procedures in our system are represented by a **FuncStorage** object, which has the necessary properties for an RPC. The relevant properties for the binder are: *function name*, *function argument types*, the *list of servers* (represented by a string for the host, and an integer for the port) that have registered this function.

The database is represented by the class `FuncDatabase` and contains:

- A Map data structure, indexed by function name, where the values are vectors of FuncStorage

objects.
- A vector of servers that have an active connection to the binder
- A round robin index that represents the token

Function overloading

The Map data structure is indexed by function name, and if there is no corresponding key existing for a currently registering function, this means that a new entry will have to be made in the database. Accordingly, a new entry with the function name, and server info is created according to the database scheme.

If the function already exists in the map, then the corresponding list of servers is retrieved from the database, and scanned. If the server also exists, then no changes are made to the database. If the server does not exist, then it is added to the list of servers for this function.

Comparing functions

To check registering functions for equality, we implement the function `vector<int> reorderArgTypes(vector<int> & types)` . This function is used to reorder the *output* argument in the argTypes array so that two functions can be compared accordingly. This is done to handle cases where a user specifies the output argument after the first index in the argTypes array.

Two functions are considered equal iff they satisfy the following conditions:

- They have the same names,
- They have the same arity,
- They have the same output type,
- They have the same argument types,
- They have the same argument order

Note: If an argument is an array, then the length of the array is ignored.

Round-robin scheduling

`Binder::getServerRR(string func_name, int func_id)` is responsible for fetching the next server that can handle a given function. The binder maintains a token that indicates the server that last executed a procedure. When handling a client request, the token is incremented by 1, and then the binder loops sequentially over the list of servers for one that can process the given function. If a server is deleted, then the token just advances to the next server in the sequence.

Termination procedure

Client

When a client calls `rpcTerminate()` , it sends a terminate call to the binder. In the event that a single client calls `rpcTerminate()` , if there are any RPC procedures currently being executed, they exit immediately with the appropriate exit code.

Binder

The binder has a single boolean variable, **terminate**, that tracks whether or not it has received a terminate request from a client.

The binder also maintains a list of servers that it has an active connection to.

On receipt of a termination call, the binder sets the **terminate** variable to true and begins sending out termination requests to servers. After a termination request is sent out to a server, the binder removes it from the list. Once the list of servers is empty, the binder itself closes its socket, breaks its infinite loop, and then exits.

Server

Upon receiving a terminate call from the binder, a server closes all of its sockets and terminates.

Optimizations

No extra optimizations have been implemented.

Error codes

1. Init:

Warning:

1 : A server has already been initialized.

Error:

-1: Cannot find the binder info
-2 : Fail to create a connection socket to be used for accepting connections from clients
-3 : Fail to get the server address or the port number for the accepting socket mentioned above
-4 : Fail to connect to the binder

2. Register:

Warning:

1: duplicate registration, the func has been overridden on the server side

Error:

-1: the server has not been initialized.
-2: the server is executing.
-3: fail to send message to binder
-4: fail to receive message from binder
-5: registration failed

3. Execute:

Error:

-1: no function has been registered yet

4. Call:

Error:

-1: Cannot find the binder info
-2: Fail to connect to the binder
-3: Fail to send message to binder
-4: Fail to receive message from binder
-5: Fail to locate the func from binder
-6: Fail to connect to the server
-7: Fail to send message to server
-8: Fail to receive message from server
-9: Execution of func failed
-10: Execution of func crashed

5. Terminate:

Error:

-1: Cannot find the binder info
-2: Fail to connect to the binder
-3: Fail to send the termination message to the binder

6. CacheCall:

Error:

-1: Cannot find the binder info
-2: Fail to connect to the binder
-3: Fail to send message to binder
-4: Fail to receive message from binder
-5: Fail to locate the func from binder
-6: Execution of func failed
-7: Execution of func crashed

Missing functionality

All functionality has been implemented, including caching.

Caching mechanism

When a client calls `rpcCacheCall()` it sequentially checks the local database for a matching server. If a function exists, the client makes a direct request to the server. If a matching server couldn't be found, the client queries the binder for one that can service its request. If the request fails, it searches in the cache for another server that can handle it.

If the client iterates through all the servers in its local cache, and is still unable to have its request fulfilled, it requests an updated list of servers from the binder for the given remote procedure. It then caches these, and then sequentially tries servers to fulfill the remote procedure call, returning with a code indicative of the execution status.

Round robin

For RPC calls where the functions are all contained in the local cache, round robin functions identically to the non-caching method.

The special case for round robin scheduling for `rpcCacheCall()` is when the client can't find a server to handle a function in its local cache. The client must then retrieve the necessary server information from the binder. On retrieval, the servers are added to the *end* of the client's local cache, which guarantees that the token is passed sequentially in the correct order.

Example:

Servers A, B, C are started in order and have the functions:

```
A:   x(), y()
B:   y(), z()
C:   z(), x()
```

and the client makes the following requests sequentially: `x x y y z z`

The requests will be handled in the following order: `A C B A C B`