CS 454 | A3
System Manual

Sivakumaran Sivanathan - ssivanat
Talha Khalid - mtkhalid

Design Choices

Marshalling/Unmarshalling of Data

The data was marshalled – at the client side – and unmarshalled – at the server side – through serialization (bit-shifting). Before passing any data, we first pass an RPC_MSG which will indicate two key parameters. The first is the type of request (i.e a Location Request, Execute request, etc). The send is the length of the entire message so that the receiver could allocate enough memory to handle the data transfer.

Now before we can add data onto a buffer, we must tell it the size of that data so that the receiver can distinguish between different parameters on the buffer. So an integer is placed first in the buffer followed by a parameter until all parameters have been added. We then send the data over and the receiver removes the integer first, creates a variable to hold a variable of that size and then copies the argument from the buffer.

Binder Database Structure

Our database implementation is implemented as a **map** of DBEntry instances. A DBEntry instance is one entry into the Database. It contains the name of the function, the argTypes, and which server nodes the function is located at.

```
struct DBEntry{
        char* name;
        int* argTypes;
        std::queue<ServerNode> serverList;
};
```

Error Codes

The error codes were defined as follows; these error codes corresponded to:

- the server's registration with the binder
- the client's lookup of the server
- the successful execution of an RPC call

Server's Registration with the Binder
These codes indicated whether the server successfully registered its functions with the binder.

REGISTER: 1
Used to register a server's function with the binder

REGISTER_SUCCESS: 2
Used if the server successfully registered its functions with the binder

REGISTER_FAILURE: 3
Used if the server could not successfully register its functions with the binder

Client's Lookup of the Server
These codes indicated whether the client successfully located the server, via the binder.

LOC_REQUEST: 4
Used in the case when the client requests to lookup a particular function at the server

LOC_SUCCESS: 5
Used if the client's lookup request was successful

LOC_FAILURE: 6
Used if the client's lookup request was un-successful

Successful Execution of an RPC Call
These codes indicated whether the RPC Call successfully executed at the server's end.

EXECUTE: 7
A code used to execute the RPC call at the server's end

EXECUTE_SUCCESS: 8
A code used when the RPC Call execution was successful

EXECUTE_FAILURE: 9
A code used when the RPC Call execution was not successful

Termination of a Server Message

TERMINATE: 10
This code was used in the message sent to all servers, asking them to terminate

Round Robin Scheduling

Our scheduling algorithm follows a simple round-robin scheduling scheme.

Given a client request of functions - f(), g(), h(), f() - corresponding to a particular list of servers - A, B, C - each server capable of serving the request is able to do so, and is given equal priority. In the case of the first client request - f() - if A is able to serve the request, it does so. If not, the chance to service is passed on to B; if B is able to serve the request, it does so. If not, the chance to service is finally passed to C, whereby it's checked if C is able to serve the request.

To implement this in our implementation, we traverse through our vector of DBEntries.

```
struct DBEntry{
        char* name;
        int* argTypes;
        std::queue<ServerNode> serverList;
};
```

Each DBEntry contains a function signature - its name and argTypes - along with the list of servers that have that particular function.

Handling function overloading

We handle function overloading in binder::handleLocateServerRequest(). We first extract the function name. We then iterate over through all of the entries in our binder database that match the function name. As we iterate along, we are comparing the arguments and if we find two arguments of the same index that do not match, we conclude that this isnt the function we are looking for and move on to the next possible match for the given function name. If we get to the end of the database and find there are no matches, we conclude that the function does not exist and pass back an appropriate error message.

Termination procedure

Our termination procedure was very simple. In binder.cpp, handleTerminateServerRequest() sends a TERMINATE message to all of the servers. In the implementation, we iterate through the vector of server nodes and write a ‘terminate’ message at each server node’s socket.

Implementation-Specific Details

Client’s ability to locate a particular function

In *handleLocateServerRequest(int msgLength, int socket)*, the binder receives a LOC_REQUEST message from the client. It matches the client request to functions registered at the binder via the round robin algorithm. The binder returns the appropriate location to the client; the client is subsequently able to invoke rpcCall.