System Manual

Alex Tse (acytse, 20350988) and Jiayi Wang (j469wang, 20501675)

Design Choices

Binder

The binder database is implemented as a vector of registration structs. Each registration struct contains data about a procedure's name, a list of its argument types, the server address and port of one server that can handle the procedure, and a "robin value" to keep track of load distribution using round robin. This implementation has a few obvious disadvantages, chief among them is the fact robin values have to be updated on each registration struct for a given server rather than keeping track of all of a particular server's data in one place. However, this design choice was finally settled on despite its shortcomings because it's easy to visualize and understand and because we felt it posed the lowest risk regarding potential knock-on issues.

Round-Robin

Managing round-robin scheduling using the system described above is fairly straightforward. When accessing the database to retrieve server information for a given procedure, a server with the lowest robin value is selected. Every time a server is requested from the binder, each registration with the same server address and port information in the database has its robin value incremented and thus kept in sync with each other. The client side don't handle it own round robin schedule, neither do the server side. Client just call rpcExecute() whenever it got the server location, and server create thread immediately when it get a execute request. They both trust the binder for the scheduling.

Overload and Override

Given the way the binder database was implemented, overloading came "for free". Since procedure registrations are all independent of each other, an overloaded procedure would just be registered as normal since it won't match any other existing registration exactly (otherwise it would be an override not an overload). Overrides are handled naively by deleting an existing duplicate registration and replacing it with the new one.

Termination

Termination from the binder's perspective is simple. When the terminate protocol is received, the binder relays the termination to each registration in the database. This means that servers with multiple procedures registered will be sent multiple terminations, but we felt this is actually beneficial because of the additional redundancy provided. The binder does not wait for acknowledgement from the server before terminating itself.

Data marshalling

Data marshalling is fairly rudimentary in our system. Using the protocols set forth in the assignment spec as well as other standards we agreed on throughout development of the system, we were able to ensure that communication between the client, server, and binder via the library we implemented works as intended.

1. We decided that server addresses and procedure names would be limited to 128 characters including a null terminator and that addresses names shorter than this length would need to be padded.
2. For argTypes, we just send a single array of all integers including the trailing zero.
3. For args, the sender first calculates the length of the input and output arrays (single values are treated as an array with one element), then allocates memory for both of them. The args are then copied into an input buffer, output buffer, or both. The receiver also needs to calculate these two lengths based on the argTypes array it received, and use this length to allocate enough memory to receive the data. Unmarshalling the data is also done using the argTypes array, the receiver needs to split the big array into smaller arrays of the right types, and create a void** pointer to handle all the addresses. By this design we only send data once in each direction (input data only from client to server, output data only from server to client), in order to improve the efficiency of our system.

Error codes

ERROR_INVALID_BINDER_INFO = -10; // when environment variable binder_addr or binder_port are invalid

ERROR_INVALID_ADDR_INFO = -11; // when server create a socket to accept connection from clients

ERROR_FAILED_TO_BIND = -12; // cannot bind to binder

```
ERROR_FAILED_TO_LISTEN = -13; // server failed to listen

ERROR_FAILED_PORT_RETRIEVAL = -14; // failed to get server port number

ERROR_REGISTER = -15; // cannot register on server

ERROR_FAILED_TO_SELECT = -16; // server failed to select from FDs

ERROR_FAILED_TO_ACCEPT = -17; // server failed to accept clients connections

ERROR_BINDER_CLOSED = -18; // binder is closed

ERROR_NO_BINDER_AVALIABLE = -19; // cannot reach binder, maybe no environment variables or binder is down

ERROR_CLIENT_FAILED_CONNECT_SERVER = -20; // client got server's location, but cannot connect to that server

ERROR_BAD_ARG_TYPE = -21; // when the arg type is unknown

ERROR_LOC_FAILURE = -22; // when client get LOC_FAILURE from binder
```