Armand Halbert
Zudong Wu
March 7, 2014

Lab 1 Design Document

Introduction

The design of our project consists of three parts: a Bard process, a Server process and Client processes. The Bard announces scores, the Server stores and communicates scores, and the clients get scores displayed to the users. Interprocess communication is implemented entirely with *remote procedure calls*.

The remote procedure call design allowed for complete transparency in interprocess communication. To each process, it appeared as if it were calling a method defined in the program listing, rather than communicating with a remote process. This simplified the design of our project, compared with programs that use sockets and pipes.

However, there were some disadvantages to the RPC design. In order to push updates to the client, we were forced to have the client register a remote procedure call and act as an RPC server, despite being a client to the main Server. While this design did not break transparency, it did create leaks in the Client-Server-Bard abstraction. Overall though, the design worked well and allowed us to implement interprocess communication without any problems.

To address synchronization, we used the *second type reader and writer lock* paradigm. Only the server is multithreaded. The multithreaded server allow the probable highest parallelism among the bard, and clients processes. Though we know the type of reader and writer lock may cause starvation of readers, we claim this would very unlikely be a problem for our system. Because the only writer is the bard process, whose generation of scores and tallies are regulated.

Other features include:

- Locks and program logic guarantee that interaction between scores and tallies is right: the score for an event would not change after the event is flagged as ended; exact one gold, one silver (one bronze if we have the third team) would be granted for one event at the end of it.
- `pull` and `push` mode are integrated, providing users with a unified interface (client.py)
- Messages are recorded in output files (./src/client/log/*.out)
- Interactive shell in client for ease of use

Usage

1. Launch the server. `python ./src/server/server.py &`
2. Launch the bard in the background. `python ./src/bard/bard.py &`

3. Launch as many clients as necessary. `python ./src/client/client.py`
Each time you launch a client, you must change the port number for `self_port` in client_config.py.
4. Exit the client with `<CTRL-C>` and use the `kill(./src/admin_stop.sh SCRIPTNAME (e.g., server.py))` utility to end the Bard, Client, and Server processes.

To run the system over different machines, change the ip addresses in the configuration files to the relevant machines. By default, the processes will try to connect to localhost.

Design

Client

The client provides a basic shell that allows the user to request information. Options include registering, deregistering, getting the medal tally and displaying the score. It communicates with the server via remote procedure calls.

The client has two modes-unregistered and registered mode. Registered mode subscribes a client to a sporting event that is going on, with the scores stored on the server. In register mode, an RPC server runs that allows the main Server to push updates to the client. When the client deregisters, the server refuses connections. While registered, the client may unregister to stop receiving score updates, and is able to do all of the things the client can do while unregistered, such as ask for medal tally and register for other events. The client may re-register as long as the game continues, and receive updates again.

Client Commands

- `REGI <event>` - Registers client to receive updates for `event`.
- `DE_REGI <event>` - Deregisters client from `event`.
- `MEDAL <team>` - Gets medal tally from Server.
- `SCORES <team>` - Gets latest scores from Server.
- Type `<Enter>` button for commands instructions

Server

The server acts as a simple intermediary interface between the clients and bard. It stores the event scores and medal tally and returns them to the client. It can register clients for an event, and contact the client's RPC server to push updates to the client.

The server's design is simple - it is run as a Python `simplerpc` server. When a new request arrives, a thread is created to handle the request. To ensure that there are no concurrency problems, we use a series of locks to protect critical sections of code. In addition, we employ a one writer, many reader interface. There is only one thread that can write to shared memory, while many threads can read from it. This prevents threads from overwriting information.

Bard Interfaces
incrementMedalTally - increments the tally for a team and medal type
setScores - sets the scores of the specified event, and whether the event has ended

Client Interfaces
getMedalTally - get medal tally from the server
getScores - get Scores for an event from the server.

Bard

The bard process determines when the score and medal tally should be incremented. The thread that updates the score and medal tallies runs every 2 seconds (*you can change the parameter in the bard configuration file*), using random variables to decide who scores. When the game ends, medals are awarded, and when all the games are over, the Bard shuts itself down. Remote procedure calls are used for communication between the Bard and Server.

The bard acts as a thin client to the Server - it does not actually store data, but pushes updates to the server through incrementMedalTally and setScores. This simplifies the Bard's design, avoiding the problems created by sockets and multithreading.

Testing

Performance and unit tests are provided in test.py, which interacts with the remote procedure calls developed in our project. We used the script to test the correctness of our implementation. However, we found unit testing insufficient for testing programs that utilize interprocess communication and random variables. We also tested the program by verifying that each of the processes worked correctly by themselves, and observing the output.

We then tested our system on multiple machines. One problem is that we coded the ip address into our configuration files, so the program must be modified to set it up for a distributed system. The version submitted is the version with localhost as the target computer.

While latency increased slightly, we noticed little performance change when running the program as a distributed system. The overall number of clients did not affect latency. In our deployment, we did not notice any differences in latency with regard to latency between `pull` and `push` mode.

Number of Clients

	6	10	14
Local	1 ms	1 ms	1 ms
Distributed	8 ms	7 ms	7 ms

Table 1. Latency of various setups of our system

Conclusion

Some possible improvements to our program include supporting any number of teams and events (we could add teams and events in the configuration file), dynamic distribution of configurations (we could achieve this by letting the bard, clients processes explicitly request the server for port configurations immediately after an initial connection), using sockets to implement communication from the server to the client, etc. Overall, this project was a great introduction to the challenges of building distributed systems. Some minor flaw is due to we integrate the `pull` and `push` mode in the client together, the message obtained from `push` mode would affect the interaction through command window; if you uncomment prin command in the `client.py`, and start with ./src/client/client.py 2> /dev/null, the issue will be relieved to the level that only newline characters returned by the RPC server of the client will show up (I failed to find methods to address this since the RPC class is encapsulated, and there is no related document). Though, all functions still work ok. We hope we can address this issue in the next version (e.g., using socket instead of RPC, we can easily direct the obtained pushed messages to an output file).