Armand Halbert
Zuodong Wu
May 7, 2013

Lab 3 Design Document

Introduction

Our design is based of our design in lab 2. It consists of multiple frontends (there are two in our code, but more than two frontends can also be supported), a database backend, a bard, and a number of clients. The ***bard*** sings to the database, telling it to update scores based on events at the olympic games. The ***database*** acts as a simple file with read/write methods. It is contacted by the ***frontend***, which requests information to pass to clients. The frontends are distributed and load balanced to improve performance. In this lab we implement ***caching on the frontend***, as well as ***fault tolerance*** to handle problems in case a ***frontend crashes***. Clients connect to the frontend, where they request score updates from the server.

We used ***Remote Procedure Calls (RPC)*** to implement message passing between the frontends, database, and clients. The RPC calls allowed us to pass and receive messages from each process without having to manage sockets or pipes. RPCs acted as if calling a normal method, helping to avoid bugs in the system through transparency.

There are 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 a 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-Database abstraction. Compared to using JSON and sockets to communicate, the RPC design does not scale well. Overall, though, the design worked well and allowed us to implement interprocess communication without any problems.

Usage

Run by navigating to each directory in `src/integrated` in this order: `frontend1`, `frontend2`, `backend`, `client1`, `client2`, `client3`, `client4` and `bard.` Run with `python <filename>`, and use kill to crash a frontend. We recommend using a terminal multiplexer such as tmux or screen to run the application.

Example:

```
cd frontend1
python frontend.py &
cd ../frontend2
python frontend.py &
etc...
```

Caching

Caching is implemented in each of the frontend servers, with one of the frontends elected as the master server that the bard connects with. Each record is stored as a single list and if the cache ***invalidates*** the record, the record is set to ***None***, causing a ***cache miss***. When a cache miss occurs, the frontend contacts the database to get the latest scores. A global variable in the backend is used to set the cache mode. An RPC call is used by the frontend to get the cache mode. If the master frontend fails and comes back online, the caches of all frontends are invalidated to ensure that the latest scores are updated rather than serve incorrect scores.

Push based caching is controlled by a master frontend server. We initially decided to use the database as the master server instead of a master frontend since the database is already responsible for updating the master record of scores, and can send cache updates as soon as new scores are written. However, we found that this would require duplicating the database. Therefore, we switched to electing a master frontend via the one that has the highest process id. The master uses a RPC call to all of the frontends to signal that the cache should be updated, sending the relevant scores along with it. The master updates its scores by telling the database it is the master server, and receives push notifications when scores are updated by the bard. This avoids cache misses, since the relevant scores are sent along with the invalidate message.

In ***Pull*** based caching, each front end server is responsible for managing its own cache. ***Polling*** is used to ***periodically*** contact the database and request the latest scores. Polling intervals are based on the popularity of the sport - the more requests for a score, the more often the database is polled to check if the scores have been updated (although it is direct to implement such dynamic scheme, we only implement a scheme with a fix pre-set polling frequency). This avoids cache misses, but scores are more likely to be stale compared to push based caching.

To avoid stale scores, the Gauls would probably prefer push based caching, since it ensures scores are updated as soon as they are changed.

Fault Tolerance

The system also provides tolerance for the event that a frontend server crashes. If ***clients*** are unable to contact a frontend server they are connected to, they search for a new frontend to connect to. The new frontend communicates with the original server periodically; when it finds the crashed frontend revives, it notices the clients to reconnect to their original frontend servers. This does not perfectly redistribute the balance, since new clients could connect to the frontend during that time between the crash and restart. However, since the new clients will be rebalanced across the remaining frontends, it is unlikely the new balance will be significantly upset by a frontend crashing. If there is an imbalance, the server with the smallest load will take on new clients until the balanced is restored.

When the master fails, a new master is elected by the other processes (frontend servers); the bard would be always guaranteed to only connect with the master frontend. In the case a new master is selected, cache results are invalidated for all frontends to ensure correctness.

Testing

We first tested the push based caching system. We tested this by connecting clients to the front end, and requesting scores. We then disconnected the database to ensure that cache results were being provided, and not just forwarding requests to the database. The cache correctly served the requests, and waited patiently for the master frontend to push updated scores. Stale data occurred infrequently during testing, mostly due to the master frontend crashing during fault tolerance tests.

For pull based caching, we increased the poll time to long enough to ensure there would be a period of stale scores. We then lowered the poll time so that scores would be as fresh as possible. We then checked the value of the scores against the scores in the database and ensured that the cache was working. We observed that there was about ***2ms*** of stale data on average, using our polling rate.

Testing fault tolerance involved shutting down arbitrary frontend servers with the ***kill utility*** (integrated/admin_stop.sh) and seeing how the load was handled. We used the output to determine what servers were handling the load. We also used the output to verify the load was evenly distributed across the remaining frontends. We ***then*** brought frontend servers back online to test if the load balance correctly reasserted itself. We verified that the system handled load balancing correctly, and could recover from failure.

The Kill Utility

```
integrated/admin_stop.sh backend          (killing backend)
integrated/admin_stop.sh frontend         (killing all frontends)
integrated/admin_stop.sh client           (killing all clients)
integrated/admin_stop.sh bard             (killing bard)
integrated/admin_stop.sh all              (killing all processes)
```

Performance

Overall, performance was similar to what we found in the previous two labs. We found that performance was better than our lab 2 when the system was distributed. The new version should also scale better, since caching is implemented and improved performance. The new version is also more reliable, since a frontend crash will no longer disconnect a client.

Number of Clients	6	10	14
Local	2 ms	2 ms	2 ms
Distributed	12 ms	20 ms	19 ms

Table 1. Latency of various setups of our system

Conclusion

This was a valuable exercise in studying caching and providing fault tolerance for systems, as well as organizing one project so that it could support arbitrary improvements in the future. We also learned how to balance server loads even in the face of possible crashes.

One improvement would be to solve the Byzantine Generals problem, which provides protection against memory corruption by Roman spies. This would be implemented by implementing four frontends, and corrupting one of them.(a byzantine fault). Since only one process will be corrupted by a spy, the two-thirds requirement of the byzantine generals is passed, and the processes can come to an agreement on what the scores are, even if the master is corrupted.