Jessica Newman
Aaron Schein

Design Document for the Olympic Games Server/Client Architecture

File Structure

The system is broken up into the following classes:

- ObelixServer.py

Handles all communication with the client, and provides the interface for the remote procedure calls. The ObelixServer is a subclass of an AsyncXMLRPCServer.

- ScoreKeeper.py

Serves as an intermediary between the OlympicEvent and Team, and the ObelixServer.

- DatabaseServer.py

Handles all communication from the two front-end server processes to the database. The DatabaseServer is a subclass of an AsyncXMLRPCServer.

- DatabaseManager.py

Writes all scores to a database called scores.db, and retrieves results back for the servers.

- AsyncXMLRPCServer.py

Handles leader election and time synchronization. Each server also has a vector clock which it uses to count incoming requests.

- PygmyDotCom.py

Serves as a middleman between the two front-end servers and the clients (and Cacofonix). This class performs load-balancing between the two front-ends by randomly choosing a server to field the request.

- StoneTablet.py

The tablet polls the server occasionally for updated information about the teams and events in which it is interested.

- Cacofonix.py

Cacofonix periodically updates the server with scores for events, and medal tallies for teams. In server-push-mode, when Cacofonix issues an update to the server, an update is immediately issued to all clients registered for every event and team that was updated.

Caching

Each front-end server has its own in-memory cache. Data for an item is added to the cache whenever a StoneTablet requests data for that item. There are two modes of cache consistency, and they work to various levels of effect in our system.

1. Pull-based consistency: In pull-based consistency, the front-end server polls the database for updates at regular intervals. Our refresh-rate is relatively short, so very little stale data will be seen, and scores will never be more than a few seconds out of date.

2. Push-based consistency: In push-based consistency, the cache is updated whenever a score from Cacofonix is registered for that event or team. The data that was in the cache before the update is now out of date, so it is invalidated. When a StoneTablet makes another request, the

request will have to make the round-trip to the database. While this would seem to result in no stale data, for a system in which multiple machines can receive updates, as we have - (our infrastructure is built so that there can be arbitrarily many front-end servers) – it is possible, however unlikely, that one front-end server will rarely or never receive updates from Cacofonix. If this is the case, then the cached data may never be updated. One solution to this issue would be to have one front-end inform the other when its cache needs to be invalidated, but this results in more network communication, latency, and possibility of failure, which is exactly what we tried to avoid with caching in the first place. For this reason, I argue that a push-based approach is not sufficient for a distributed system that can receive updates at multiple terminals. The cache must be cleared occasionally in the same way that it is in pull-based consistency.

With caching, our system has an average latency of 0.204504380226 seconds per request. Without it, the average latency is 0.389653215408. So, caching nearly halved the response time of our server. This was calculated over the course of 50 requests.

Fault Tolerance

Our system of fault-tolerance works as follows:

For every request made to the PygmyDotCom frontend, (by either the StoneTablets or Cacofonix,) the system generates a list of active frontend servers by checking each of the servers for responsiveness. Then, one of the available servers is chosen at random to service the incoming request. This system is more robust than using a heartbeat for responsiveness because with the frequency with which the servers are requested, even a very frequent heartbeat could result in an outdated list of active servers and cause a crash. By populating a list of active servers with each request, there is no possibility of failure, and as soon as a server is up again, it resumes servicing requests.

Synchronization

We do allow asynchronous access to the database, though concurrent writes of medal tallies and scores, and reading scores and tallies while these are being modified is not allowed. Otherwise all actions of the servers are asynchronous.

Networking

This system is fully extensible to multiple machines. To run on multiple machines, you will need to know the external IP address of the machine that each of your servers and your database server process is running on. These processes are required to know the IP addresses and ports of one another. In the future, we would like to implement a system whereby PygmyDotCom can enter new servers into our system by informing all active servers about the new server's port and IP, so that this does not need to be determined from the start.

Security

Both Cacofonix and the ObelixServer have knowledge of a “Cacofonix password” which allows Cacofonix to modify important data. Any client attempting to call increment_medal_tally or set_score will be unable to do so without the password.

Test cases run

Information about what is being modified in the cache is displayed in the terminal. Whenever stale data is discovered, the server prints information about what data is being invalidated. In pull-based consistency mode, the server prints information about which items are being invalidated, if any.

We tested fault-tolerance by manually crashing one or both front end servers with a KeyboardInterrupt, then bringing them back up. In any situation, the system does not crash.

Run Instructions

To see the code in action:

Open 7 terminal windows, run the following commands in each (look below for extra details on each):

> 1) python DatabaseServer.py --uid=0 --port=8000 -l
> 2) python ObelixServer.py --uid=1 --port=8002 -l -c **This will run in pull-based mode**
> 3) python ObelixServer.py --uid=2 --port=8003 -l **This will run in push-based mode**
> 4) python PygmyDotCom.py --port=8001 -l
> 5) python Cacofonix.py --serport=8001 -l
> 6) python StoneTablet.py --port=8005 --serport=8001 -l
> 7) python StoneTablet.py --port=8006 --serport=8001 -l

1) Initializes the SQLite database and DatabaseServer/DatabaseManager. If you let this set awhile (before continuing to the next step), you'll see it searching for other processes.
2) Starts-up the first front-end server. Soon after it starts, it will start an election with the DatabaseServer. The outcome of the election will be printed in both terminals.
3) Starts-up the second front-end server. It will ask who the time server is and acknowledge.
4) Starts-up the load-balancing middleman, Pygmy.com.
5) Starts-up Cacofonix and begins to simulate updates to the Olympics.
6) Starts-up a client SmartStone, and immediately begins pulling updates.
7) Starts-up another SmartStone.

To run non-locally, the servers will have to know about eachother's locations. The following example uses all the necessary flags but still sets everything to run locally. To run non-locally change the IP addresses from localhost. Further details below.

1) python DatabaseServer.py --uid=0 --port=8000 --xhost=localhost:8002 --yhost=localhost:8003
2) python ObelixServer.py --uid=1 --port=8002 --dbhost=localhost:8000 --zhost=localhost:8003
3) python ObelixServer.py --uid=1 --port=8003 --dbhost=localhost:8000 –zhost=localhost:8002
4) python Cacofonix.py --serip=localhost --serport=8001

5) python StoneTablet.py --port=8005 --serip=localhost --serport=8001 -l
6) python StoneTablet.py --port=8006 --serip=localhost --serport=8001 -l

1) xhost, yhost are the addresses of the two frontend servers given by IP:PORT
2) dbhost is the address of the database server. zhost is the address of the other frontend server.
3) Same as 2.
4) Add serip (server IP) which was defaulted to localhost when running locally.
5) Same as 4.
6) Same as 4.