Jessica Newman
Aaron Schein

Design Document for the Olympic Games Server/Client Architecture

File Structure

The system is broken up into the following classes:

Server-side:

- ObelixServer.py

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

- ScoreKeeper.py

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

- OlympicEvent.py

Keeps knowledge of the current score for each event, and the clients registered to receive updates for each event. (Server-push mode only.)

- Team.py

Keeps knowledge of current gold, silver and bronze medal tally for each team, and the clients registered to receive updates for each event. (Server-push mode only.)

Client-side:

- StoneTablet.py

Two modes: Client-pull and server-push mode. Each stone tablet is interested in some certain random selection of teams and events. (As of now, the teams and events are provided as a constant list.)
In client-pull mode, the tablet polls the server occasionally for updated information about the teams and events in which it is interested.
In server-push mode, the tablet registers itself with the server, then provides its own RPC interface which the server calls whenever an update is received on one of the tablet's events or teams of interest.

- 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.

API

ObelixServer.py provides the following RPC interface:

- get_medal_tally(team_name): StoneTablet can poll to receive the medal tally for a particular team name. The ScoreKeeper will then ask the Team object associated with that team_name to send its current score, and pass it along to the client. If the client miscalls this function with a wrong team_name, an error string will be sent back.

- get_score(event_type): StoneTablet can poll to receive the current score for a given event. The ScoreKeeper will then ask the OlympicEvent object associated with that event_type to send its current score, and pass it along to the client. If the client miscalls this function with a wrong event_type, an error string will be sent back.

- increment_medal_tally(team_name, medal_type, password): Cacofonix calls this function to increment the number of gold, silver, or bronze medals for the given team. An error string will be sent back if miscalled.

- set_score(event_type, score, password): Cacofonix calls this function to update the score for a given event. An error string will be sent back if miscalled.

- register_client(client_id, events, teams): Here, client_id is an (IP address, port) tuple, events is a list of events and teams is a list of teams that the client would like to be registered for. A client running in server-push mode will call this function to register with the server. In the case of a wrong client_id, nothing will be registered. If there are one or more invalid events or teams in the events or teams list, the client will be registered for the valid ones, and an error string will be sent back for the invalid ones.

- push_update_for_event(clients, event_type): Pushes the current score for the given event_type to all clients registered for that event. It does this by calling print_score_for_event(score) on StoneTablet.

- push_update_for_team(clients, team_name): Pushes the current score for the given team name to all clients registered to that team. It does this by calling print_medal_tally_for_team(medal_tally) on StoneTablet.

StoneTablet.py provides the following RPC interface (server-push mode only):

- print_score_for_event(score): Receives a score and displays it.

- print_medal_tally_for_team(medal_tally): Receives a medal tally and displays it.

Synchronization

ObelixServer.py and the entire server-side architecture is written to be entirely asynchronous. However, we do have locks on important data structures to prevent data inconsistency. We have achieved this synchronous behavior through the use of code that wraps around each function. There are two different wrappers: One acquires the lock, executes the function, then releases the lock, and one that spin waits for the lock to be released, then executes the function. I am sure this code could be improved. Our spin-wait could be replaced with something more scalable. One advantage of our current synchronization technique is that we could theoretically handle an arbitrary number of criers. There are four critical data structures and thus four locks:

- medal_tally lock: This lock is acquired whenever Cacofonix calls increment_medal_tally. While this lock is acquired, any thread attempting to access get_medal_tally must block until the lock is released.

- score_lock: This lock is acquired whenever Cacofonix calls set_score. While this lock is acquired, any thread attempting to access get_score must block until the lock is released.

- event_client_list_lock: This lock is acquired whenever a StoneTablet wants to register in server push mode for a specific event, and an OlympicEvent's clients list must be modified. Any attempt to access push_update_for_event while this lock is acquired must block until the lock is released.

- team_event_client_list_lock: This lock is acquired whenever a StoneTablet wants to register in server push mode for a specific team, and a Team's clients list must be modified. Any attempt to access push_update_for_team while this lock is acquired must block until the lock is released.

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 your server is running on. Then you can provide this as a parameter in the bash scripts start_olympics.sh and start_clients.sh. You must also flip the 'run locally' flag to False in start_server.sh and start_clients.sh.

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.

Testing

Both the process updates (Cacofonix) and the clients can be set to testing mode using a simple flag (explained below). This allows us to read the output logs from the server, Cacofonix, and the clients to confirm correct behavior.

Performance

See latency_push.png, latency_pull.png and comparison.png.

In the case of the server-push architecture, we varied the number of clients from 1 to 50 with various intermediate values. The average latency is plotted in latency_push.png. We see that it rises sharply but appears to taper after 40 clients.

Run Instructions

[Instructions for running bash script]

First, start up the server:
bash start_server.sh

Then, start the update process which will simulate the Olympics and Cacafonix's updates:
bash start_olympics.sh

Finally, fire up the clients:
bash start_clients.sh

NOTES on the .sh files:

There are various flags in each of the three bash scripts that allow you to configure each process.

In start_server.sh, specify whether to host the server locally or not using the ``run_locally" flag. You can also designate a port for the server other than 8000.

In start_olympics.sh, you must specify the ip and port of the server. If the server is being run locally, then ``localhost" is the correct input. Otherwise, you must find the IP of the server (not including http://) and set the server_ip flag to that. The same goes for the server_port flag (if something other than 8000). You can also set the Cacofonix's port to be something other than 8001. There is a flag called ``mode". When ``mode=random", the updates to the medal tallies and scores of events are all stochastic (as is when they occur). When ``mode=testing", deterministically at regular intervals, Gaul wins a gold medal (medal tally update) and Gaul leads the event Stone Curling by 1 (event score update). No other updates occur. This allows us to check that the system is behaving correctly all the way through the client updates. Finally, there is a flag called ``rate" which controls how frequently updates occur. When ``mode=random" updates occur randomly between 0 and ``rate" seconds. When ``mode=testing" updates occur at exactly "rate" seconds apart.

In start_clients.sh, you also must specify the ip and port of the server in the same way as start_olympics.sh. Like in start_server.sh you can also choose to run the clients locally with the ``run_locally" flag. There are two for-loops in start_clients.sh. The first one will spawn N clients in client-pull mode. The second one will spawn M clients in server-push mode. The same server and

update process can serve both client-pull and server-push clients concurrently. The flag ``arch" (for architecture) is first set to pull before the first for-loop, then set to push. Port numbers for each client are generated as a sequence starting from 8002. If that port is unavailable, the client will randomly select a new port from the range [8002,9000] randomly until it finds an open one. Like start_olympics.sh, there is a ``mode" flag that can be set to random or testing. In random, the teams and events that each client subscribes to is stochastic. In testing, every client subscribes to Gaul and Stone Curling only.

NOTES on logging:

Each of the three types of processes (server, update process, and client) log their output to a different file. In the case of the server and Cacofonix (the update process) there is only one log file called log_server.txt and log_cacofonix.txt, respectively. In the case of clients, a two folders named client_logs and latencies are automatically generated. Each client generates two logs: one in client_logs, which will log the output of the client (e.g., updated scores) and one in latencies, which will log the latency in seconds of every pull or push (depending on the client's mode). Both the client_log and the latency log are named for the port of the client (e.g., 8004.txt).

To obtain an example of output, simply run (using the above instructions) and inspect:

log_server.txt ← Server output
log_cacofonix.txt ← Cacofonix's output
client_logs/PORT.txt ← where PORT is some port number of a client. Note that clients in pull mode will exhibit slightly different outputs than those in push mode.