Software Engineering Large Practical: SquareStats

Rikki Guy - s1215551

December 18, 2014

If you are starting here, I recommend running the installation instructions in README first, as they take a very long time!

1 Introduction

I have created a stats collection system, to gather and present data from the multiplayer first-person shooter AssaultCube, dubbed SquareStats. All the core parts of the proposal have been successfully implemented, making the system able to collect data from modified AssaultCube servers and store them to a remote database in real time. The system consists of four server daemons communicating with each other in a relay fashion, taking in-game frags through to web-based rankings.

I am largely content with the state of the project, as the basic functionality is implemented and works. game can be played and recorded with the rankings updated immediately after the game. I would like to have implemented more features in the time, but underestimated the amount of time the project would take and wasted time on code I would later throw away. Some components have a more complete feature set than others, however the structure of the project makes it very easy to extend with additional functionality.

2 Structure

The system is composed of five components: The modified AssaultCube server, an AssaultCube server manager, the statistics collector, the database access layer, and the web app. These are named the game server, the Manager, the Collector, the Schema and the App, respectively.

All of the components except the Schema are run as daemons. They communicate by passing JSON-encoded messages over ZeroMQ sockets, a protocol that is completely network transparent. This allows each daemon to run on separate machines or containers if desired, an increasingly useful property in today's world of virtualisation.

Originally all the components were contained in a single repository, and development of each on a different branch. But after a while I thought it is easily feasible that someone would want to run some but not others, so split them out into separate repositories. This also made much more sense from a source control perspective, as I then started to put branches to better use, i.e. per feature of the component, rather than having large branches in one large repository.

2.1 AssaultCube Server

The assaultcube server modifications are written in C. While the server itself is written in C++, the style of code used is much closer to that of C, and I am familiar with C but not C++. I have kept the additions separate from the main body of source code, placing new files in a separate directory, to prevent the AssaultCube developers' anarchic style of code causing problems with my own. Instead, I opted to create something akin to a library, and exposed an interface of hooks, to be called at the relevant sections in the server.

2.2 Manager

The manager is the daemon than connects to, and receives events from, the game server. The manager has a plug-in architecture, in that all of it's effects are defined by plug-ins. When a new Manager is created, the plug-ins to load are passed as a parameter, so can load plug-ins from any namespace or distribution. When events are received and processed, it will call the relevant on_<event_name> method on every plug-in that supports it. This not only allows arbitrary plug-ins to receive events, but plug-ins to receive arbitrary events, i.e. someone could modify the server to feed more events, and create a plug-in to deal with them, without ever having to touch or even look at the Manager source code, nor any other plug-in.

I placed the Manager in the AC:: namespace rather than SquareStats::, as I see it can be used in many more situations than this system, and plan to release it as a separate public distribution. I also plan to add an RPC-like interface between the Manager and the game server, so that the manager will have a way of controlling the server, as well as just receiving events. This would turn the Manager into a full-fledged server plug-in system, allowing people to easily create server mods in a reusable way, in a higher-level language, and without having to wade through the source of assaultcube's server.

2.3 Collector

The Collector consists of two parts: the Collector server receiving game data, and the plug-in for Manager, which collects and sends the game data to Collector at the end of a game. The reasons for waiting until the end of a game to send data are twofold. First, to reduce the load on the database server, as this way it only has to deal with one batch update (per server) ever few minutes, whereas recording all events as-and-when they happen would hit the database with writes many times per second. Secondly, the database schema stores game records, so a game cannot be recorded until it is completed. It could be possible for partial games to be kept, e.g. by allowing the end time to be NULL, however I don't think this would be a good idea. I think the database should try to "keep sync with reality" as much as possible, so that an incomplete record is worse than simply dropping the data. The stats we are recording are not mission-critical, so an odd dropped record with have no real consequence.

2.4 Schema

The Schema is almost entirely automatically generated. I initially wrote a database schema in SQL DDL, and when I started using the DBIx::Class ORM, found the DBIx::Class::Schema::Loader module to generate the ORM class files, rather than re-write the schema with a new syntax. This is why I don't have any tests for the ORM classes, as they should be covered by the tests of the module(s) that generated them. At the very least, these are the bottom of the pile.

The Schema is included in it's own distribution, because it does not belong wholly to either the Collector or the App, but is shared between the two, so I spun it out into it's own distribution with the Collector and App depending on it.

2.5 Web App

The web app is the last of the components in the chain, displaying information to the end user. Currently, only a simple leader board showing the top ten scorers is show. Score is calculated according to the in-game formula $score\_total = 2 * gibs + frags - suicides$. I plan to implement make per-Game and per-User (profile) pages, clan detection and score board, and Live games, using a plug-in to Manager, websockets and HTML5 to display live events.

The design used is borrowed from http://purecss.io/, and uses the Pure css framework. I use a wrapper template, and separate css/javascript files for all content, so the style for all pages on the site can be changed from one location.

I initially implemented user authentication where a user could link their account to a particular nickname. However, in the game there is no registration system, anyone can type /name <newname> to change their name to anything, even if someone on the same server already has that name, then there are just two (or more) people with the same name. This is also how it is valid for a kill, which is not a suicide, but has the same name for killer and victim. To this end, it is unrealistic to link an account with a single nickname, and so I decided to remove user authentication.

Due to my underestimating the amount of time it would take to implement the core features, and wasting time implementing now thrown-away functionality, the web app has suffered most when it comes to feature loss, hence only being a single page.

3 Testing

My main testing strategy has been to create a socket to simulate the remote server, and send or check messages on it. Particularly difficult, was the testing for AC::Manager and AC::Manager::Plug-in::Base, as a plug-in cannot be instantiated without the Manager, and the Manager cannot produce output without a plug-in. To solve this, I had to create a mock Manager/plug-in to load and test the other.

I estimate that my tests cover about 50% of cases of the code, however more work is required to cover the remaining test cases.

4 Future Work

The first priorities for future work lie in increasing the test code coverage, and the functionality of the web app. Adding support for new modes is a must, and supporting similar modes to osok, i.e. no teams no flags, supporting teams or flags will require minor changes to the database schema and message attributes.

One problem I faced, was lack of validation for the JSON messages, where they would arrive with missing or too many attributes, causing strange errors. In the future I should add tests that validate the messages sent and received, possibly using Data::Validate::Struct.

5 Conclusion

Overall I am happy with the progress that has been made. There is still a lot of work to be done to make this into a more stable and full-featured system, and I hope to continue it. I particularly like the flexible design of the Manager, and the possible future direction it could take on it's own. The project as a whole too, with a little fine-tuning, could easily be put to real-world use.