Project Report 4 – Outcome.

Authors:

Andreea Babiuc – CID: 00567318

Chris Dillon – CID: 00553146

James Elford – CID: 00549198

Andrew Gurden – CID: 00552321

Simon Evans – CID: 00559960

***Abstract**: Reef is a Department of Computing (Imperial College London) 3rd Year Group Project. The brief was:*

"This project will design and implement a GUI for use with an existing, albeit low-level system that automates the deployment and execution of large-scale distributed experiments. The goal is to provide a user-friendly experimentation experience for the engineer. "

That existing low-level system was Vazels – a Masters project by Angel Dzhigarov. In this report we will first give some introduction to the Vazels platform, and set out the specific aims for the project, then go on to discuss the technologies we used to deliver a thin-client graphical interface for remote users through a Web Application. Some discussion is given to the choices of technology, both those over which we had a choice (such as Google Web Toolkit and RESTlite), and those we did not (such as Google's protocol buffers). The third section of this document discusses some of the software engineering aspects of the project, both specific (problems with individual technologies), and general (lessons on collaboration, testing, and design). We end the document with an outline of the successes of the project, and ways in which it could be taken forward in the future.

Throughout the development process, the team aimed to utilise open-source technologies, and ensure that work could be picked up by 3rd parties – the final code repository is hosted publicly. The project as a whole is aimed at an audience that is technically competent and willing to investigate new tools; this report is written to be accessible to the same audience, although less technically-minded readers should not find it challenging.

Reef - a graphical front-end to the Vazels platform

We present Reef - Reef is a graphical user interface to the Vazels distributed testing system. Vazels was a Masters project by Angel Dzhigarov, designed to enable developers to carry out scalable tests on their distributed systems. In his own words[1]:

> It is a tool that helps automating experiments with distributed systems. It provides "the infrastructure" to the developers, so that the latter can concentrate on the experiment specification, rather than how to execute the experiment.

We will give a technical background on the Vazels system in a later section, but for the moment it suffices to know that it runs from a command-line interface in a Unix-based environment, and is designed with remote test-beds such as PlanetLab[2] in mind. A running experiment consists of a Control Centre (the User's main point of contact), and potentially many other machines which connect to it. Reef provides a more user-friendly way to harness the power of Vazels, by allowing users to access the system using a more intuitive GUI, provided via a web interface, from any web-enabled platform (although the initial - brief - setup requires the user to have a functional SSH client).

Reef is designed to be usable by any developer of a distributed system with a basic knowledge of the concepts involved in using the Vazels tool. The documentation provided with Reef assumes this basic level of knowledge, although attempts are made to simplify things as far as possible for new users.

Motivation

The motivations for building a GUI interface to Vazels instead of continuing to use the command line can be broken into two broad statements:

- **Command line tools are more difficult to configure.** Whilst this is something of an over-simplification, we invite the reader - by way of example - to consider their choice of operating system. Those using either Microsoft Windows or Apple OS X will be very used to using a graphical user interface to perform nearly every task on their computer. Now we invite the reader to consider configuring a Linux computer without the use of a window manager such as Gnome or KDE.
    - Vazels requires a remote user to be confident using a command-line, regardless of their background. In addition to all input to Vazels (which includes lengthy sequences of parameters on every call), they must

[1] http://vazels.org front page, 3 Jan 2011

[2] http://planet-lab.org

also be confident using both SSH and SCP. They must also be confident setting up security configurations on a remote machine (we will come back to this later).

    - Mistyping a single character on a command-line interface can invalidate a whole instruction. By providing a graphical interface for the user, we can avoid such frustrating errors and allow the user to focus on the task at hand.
    - Providing a GUI removes the “What next?” stage from the user experience - it is possible to make it clear at every stage how the user should proceed. Traditionally, command-line tools have expected user to read manual pages or seek online tutorials, but this frustrates users, and many give up trying before they produce any results.

- **Similar tools already provide GUIs.** Vazels is only one among many distributed testing platforms for users to choose from (for a list of tools that PlanetLab users are encouraged to use, one might check http://planet-lab.org/tools). It would be optimistic not to expect that for an inexperienced user the distinguishing factor may be the provision of a GUI.

Aim

We discuss the more technical key requirements of the project in the final section, entitled “Validation & Conclusions.” For now we provide only a broad overview - in short, Reef provides users with a way to harness the Vazels system from a remote location whilst avoiding the command-line. More specifically:

- Reef should provide the ability to configure and run an experiment with Vazels, using the command-line as little as practical.
- Reef should be accessible by a remote user - i.e. the user should not require physical access to the computer on which they are running Vazels.
- Reef should be platform-agnostic to the user; that is, the user should not be required to run a specific operating system.
    - Whilst it is programmed in Java, the Vazels system is limited to running in a Unix-based environment. We did not wish to impose such restrictions on the user configuring and running the experiment. This means that while Vazels itself must still run on a Linux computer, the user should be able to configure it from any platform using Reef.
- Reef should be robust: if the network fails during the initial stages of setup, or the user needs to stop half way through and return later, Reef should handle this gracefully.
- Reef may not initially provide the user with the full functionality of Vazels, but rather a significant subset (sufficient to run experiments using the basic capabilities that Vazels provides).

    - The mapping restrictions functionality is currently not implemented, meaning that users cannot specify specific hosts to be put into a particular group.
    - The “queries” and “parametrizers” features are not available directly using Reef.
- The user should be able to view the numerical outputs from their experiment and download raw data for use elsewhere.

These goals are refined by the following technical aims:

- Reef should use a Web page to deliver its interface.
    - Alternatives include X-server forwarding, or a client-side application connected to a remote server. We will discuss our choice during the technical description.
- Whilst the user must be able to view Reef from any platform, the Reef application itself is required only to run on Linux (it must have direct access to Vazels, installed on the local machine - and since Vazels is not cross platform, it would be meaningless for Reef to be), however it should be accessible by users from anywhere.
- The system should be tolerant to failures; if a connection is lost, or a server crashes, the user should not lose all of their work.

Ideally, once Reef is configured on a remote system, the user should be able to access it from anywhere - so long as they have a web browser.

Technical Background - the Vazels System

The following is adapted from our initial report (Report One: Project Inception)[3]. It describes the architecture of the Vazels system - which will illuminate the discussion on the technical details of our own system:

> The Vazels system has at its head a Control Centre, which administers several groups of remote hosts. From the Control Centre, a user can start or stop the experiment, configure a set of tests for remote hosts to carry out, and collect data from the running hosts in the network.
>
> The user can perform administrative tasks, such as configuring which tests should run on which hosts, and how many of each type of host there should be on the network. We, intuitively, call a group of hosts which are configured identically a “Group”. A Group consists of a number of remote hosts, and specifies what the hosts it contains can do. It does so by means of a “Workload,” which tells the system about snippets of code which will be stored on the hosts of the group and named “Actors.”

[3]All project reports are available in the “docs” directory of our main submission

Broadly speaking, the hosts fall into two categories: Vazel machines, and "SUE" machines. Vazel machines hold user-created tests, which are invoked at run-time and will report back information. SUE stands for "System Under Experiment" – it is the system that the user is trying to test out using the distributed network. It is worth noting that a host can be both a SUE machine and a Vazel machine – that is, it is both running the System Under Experiment, and running a set of user-created automated tests.

Once the user has prepared the experiment, they are given a probe. The probe contains enough information for a machine to connect to the Control Centre. It can then determine which group it will join and download the necessary actor and SUE files that it needs to execute.

In the Vazels system, a user must utilise the command line to start the Control Centre, specifying the names and sizes of all the groups, as well as other configuration options (for security, and regarding the mechanisms that Vazels uses for internal communication[4]).

The Vazels concepts can be visualised by the following graphic from the Vazels website:

[4]Specifically, Java RMI and Siena

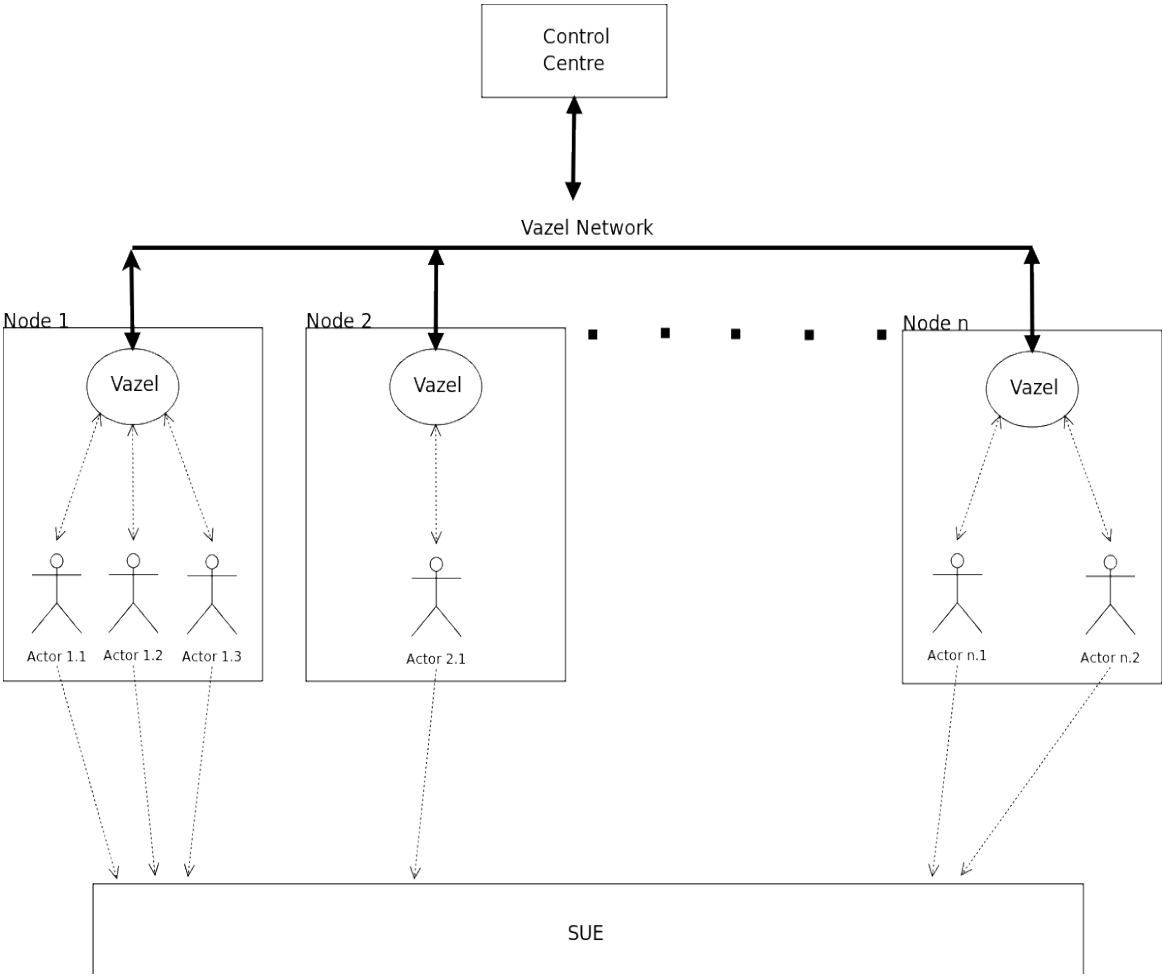

Fig. 1. *Outline of Vazels experiment execution model*[5]

[5]Reference: http://vazels.org

Technical Description

The Technologies

We should first discuss some of the technologies we make use of in Reef, before going on to explain how these components are integrated to form a coherent system (and user experience).

As has been discussed so far, Reef provides a web-interface for users. This comes in two parts: a web-page (viewable from a web-browser), and a RESTful[6], single-user back-end. The web-page with which the user interacts is the "Client-side" code in Figure 2 (in "Setting up the Experiment – Behind the scenes"), and is written in Java, using the Google Web Toolkit[7] (GWT) to convert it into a browser-optimized JavaScript application. The RESTful server is implemented in Python on top of the RESTLite[8] project (an open-source library designed for rapid prototyping of RESTful applications).

The Vazels binaries are provided with the Reef Project, with Vazels and RESTLite being licensed under the GPLv3 and LGPL respectively. With this in mind, our project is also entirely open-source.

Why a web application?

Initially, we planned to implement our GUI in either Java (using a SWING GUI), or C++ (using a wxWidgets-based GUI). However, since one of the primary use-cases for Vazels is on a remote system, we would need a GUI to be presented on a client machine, while the back end of the application could run on a separate machine: we would need either to build a client application and a server, or use X-forwarding[9] over SSH. We saw the former as a viable option, but our experiences with X-forwarding were that it can often be unresponsive to the user, and is sometimes difficult to set up.

We chose not to use a client application either, as a modern web-browser is able to provide all the functionality we required from an interface, and we favoured an approach that required the user only to configure a reasonable minimum of software. By using a web interface instead of a client application (using, for instance, Java RMI), we reduce our dependencies (on the user's system), and

---

[6] A RESTful API generally provides users (client applications) with a set of URIs which can be called using HTTP calls in order to transfer state information. In fact, our server model is "REST-ish," because it stores user-state, but we feel RESTful is the easiest way to describe it as it operates in a single-user environment.

[7] http://code.google.com/webtoolkit/

[8] http://code.google.com/p/restlite/

[9] The X-Windows system (a widely-used window manager for Unix-based systems) can forward information about GUI applications to remote SSH clients.

streamline the setup process. From a personal perspective, as a team we all had at least some experience with building Java SWING applications, and we were keen to try out new technologies.

Why Google Web Toolkit?

GWT was a new platform to us all. Most of us had some JavaScript experience, and whilst we had agreed that a web application would be the best way to deliver the interface to the user, we were keen to avoid some of the pitfalls we had experienced programming JavaScript applications in the past - for example, unfamiliar scoping rules, and the difficulty in separating concerns between classes and modules (making it difficult to properly design large MVC[10] architectures). We were keen not to simply present a static page (updated dynamically server-side using e.g. PHP, in the style of a legacy web application), so we explored other ways in which we could deliver dynamic content. GWT allowed us to use the Java language - and with it, static type-checking, more reliable error handling, JavaDocs, the Eclipse IDE (for which GWT is available as a plug-in), and other powerful coding tools - to write browser-optimized AJAX[11] applications, compiled by the toolkit into HTML and ECMA-script[12]. By programming in Java, we could write cleaner, clearer code, and be more sure of its correctness before runtime. These benefits were helped by all members of the group already being very familiar with Java. GWT also gave a way to specify user-interface design in XML which was an improvement on writing imperatively-generated interfaces, as is common when using SWING.

Why RESTLite?

RESTLite was a light-weight, easy to use platform on which we could quickly prototype our most basic functionality, and soon proved a scalable solution on which we could build our whole server. We were keen to gain some practical experience using the Python programming language, so picking an existing Python library made sense for us. Python is a dynamic, clean and simple programming language which is also powerful and robust. With an expansive standard library, addressing modern concerns such as handling http requests without the need for other external dependencies (such as the ASIO library in C++[13]). This meant that we could keep our code base relatively small, and - since most Linux distributions include Python as standard - we would not introduce new dependencies for running the Reef server.

---

[10]Model-View-Controller : http://en.wikipedia.org/wiki/Model-View-Controller

[11]Asynchronous JavaScript and XML - A way for web pages to request new data without the user needing to refresh the page.

[12]Commonly known as “Javascript”

[13]A network library for C++ http://think-async.com/

What other tools were used?

Our server used a pure-python back-end (including the RESTLite library) - aside from the Google Protocol-buffer libraries, which we will come to later - but interfaced with Vazels using the command-line, piping outputs to temporary files. We would have liked to use a more sophisticated inter-process communication mechanism such as network sockets or D-BUS[14], however neither of these technologies was built into Vazels, so these options were not available to us without modification of the Vazels code base. Vazels uses Java RMI to communicate between different internal components, however the API has no public documentation, and we thought it more sensible to use an interface for which we had reference material.

To present data back to the user, we made use of two more Google technologies. On the server-end, we made use of Protocol Buffers[15] - Google's serialization tools, used by Vazels to communicate the results of experiments from different hosts to the central control centre. We translated these data into JSON format (using the same tools) so that they could be sent by the server to be displayed to the user on our web page. From there, the Google Charts API,[16] which has an interface included with GWT, allowed us to quickly develop clean charts based on this data. By harnessing this existing library, we could make use of functionality (e.g. tool-tips and data-point interpolation) that we may not have been able to develop ourselves within the time-frame.

During debugging and development, extensive use of the Firefox plug-in "Firebug[17]" was made. This enabled us to monitor our web page at runtime, keeping track of, for example, all network requests that were being sent to the server, along with the responses (which, in certain cases would contain stack traces which we used for debugging). It provided a simple way to inspect our network traffic.

When back-end testing was required before the UE elements that sent the appropriate requests had been completed, we used cURL to send direct requests to the server, including any test data in GET or POST format, depending on the request. This meant that we could rapidly prototype server-side functionality despite progress blocking in the front-end development, allowing the team to split up tasks and each make independent progress.

---

[14] http://www.freedesktop.org/wiki/Software/dbus

[15] http://code.google.com/apis/protocolbuffers/docs/overview.html

[16] http://code.google.com/apis/chart/

[17] A JavaScipt debuger packaged as a browser plugin: http://getfirebug.com/

The System

Once the Reef server is started and the user browses to the ReefFront page, usage is split into two phases: Setup and Running.

Setting up the experiment - User Perspective

In order to set up an experiment, the user needs to specify four sets of data:

- Group names and sizes,
- Actors and Workloads,
- SUE components,
- Which Workloads and SUE components each Group runs.

Each part of the intermediate two sets requires uploading files - Actors and SUE components are specified by executables (along with their dependencies), while Workloads are specified in a format documented at the Vazels website.

Because either Actors or SUE components could be made up of potentially many files, our system prescribes nothing about their structure, save that they be provided as a compressed tarball which can be extracted to complete setup.[18]

Further, Actors must be associated with Workloads - while the Vazels system itself does not require this information, it expects it implicitly: when using the tool on the command-line, a user must extract their Actors into the appropriate folders in an Experiment Setup folder, within the file system of the machine that hosted the Vazels Control Centre.

To gather all this information from the user in a way that is clear, and easy to follow, we use a tabbed interface, with the following categories:

- (1) Group names and sizes,
- (2) Uploading Actors,
- (3) Uploading Workloads,
- (4) Uploading SUE Components,
- (5) Associating Actors with Workloads,
- (6) Assigning Workloads or SUE Components to Groups,
- (7) Starting the Experiment.

The user is able to access these items in any order, however we numbered the tabs in order to encourage a linear work-flow. Careful consideration was taken with the

[18] In the most basic case, an Actor could be, for instance, simply a single Python file. However, it could potentially be an expansive module, requiring a complex file structure in order to run. Our system is agnostic to these concerns. A component of the SUE must contain at least a shell script to bootstrap the system.

order of the tabs, ensuring that once a user has completed a tab there should not be any need to return to it later (although they were free to do so if they discovered they had made a mistake). Each tab presents a simplistic interface, encouraging the user to focus on only one task at a time. We group the assignment of Workloads and SUE components to Groups in the same section because they both involve the actual assignment of work to hosts - that is, these are the two ways in which a host can be forced to execute code.

Data Validation

As always when processing user input, it was important to perform validation. There was validation included in both the client-side and server-side code. For example, workloads could not be uploaded if the name, or file, to be submitted to the server was empty, not alphanumeric or if the filename did not end ".wkld". This form of validation was used wherever possible, to ensure the user entered data appropriately into the fields, and feedback was returned so that the user could correct their input.

The content of uploaded files was not inspected, however, and if an unauthorized user were to gain access to the system, this would allow them to execute arbitrary code through Vazels.

Setting up the experiment - Behind the scenes

We use an MVC architecture, shown broadly in Figure 2, separating control classes from the GUI, in the Managers section of the diagram. These Managers wrap the client-side data with logic to communicate the changes to the server, and to provide access to data set by one part of the UI to others (for example, section (5) above needs to know about all the data set in sections (2) and (3), but it does not need to know about the way in which that data is displayed). They also handle the task of keeping data in sync with the server, and any caching tasks.

Once data is set in the managers by user-input, it is communicated to the server using AJAX requests. These requests are routed by the RESTLite system on the server (by pattern matching the requested URI) to an appropriate Handler which could deal with the request, and return data or spawn subprocesses to carry out work if appropriate.

In general, for each item of the configuration there was a single handler on the server-side. Handlers for which incoming data would affect the state of other server-side components properly propagate changes, rather than relying on the client to update all potentially affected server state whenever information changes.

The user finalizes setup in section (7), and at that point the Vazels system is started and all configuration options are applied. The reason we apply all configuration options at once, rather than as they are sent to the server, is that certain items

(such as the number of groups, and their sizes) need to be specified on the command-line when Vazels is launched, and cannot be changed afterwards.[19]

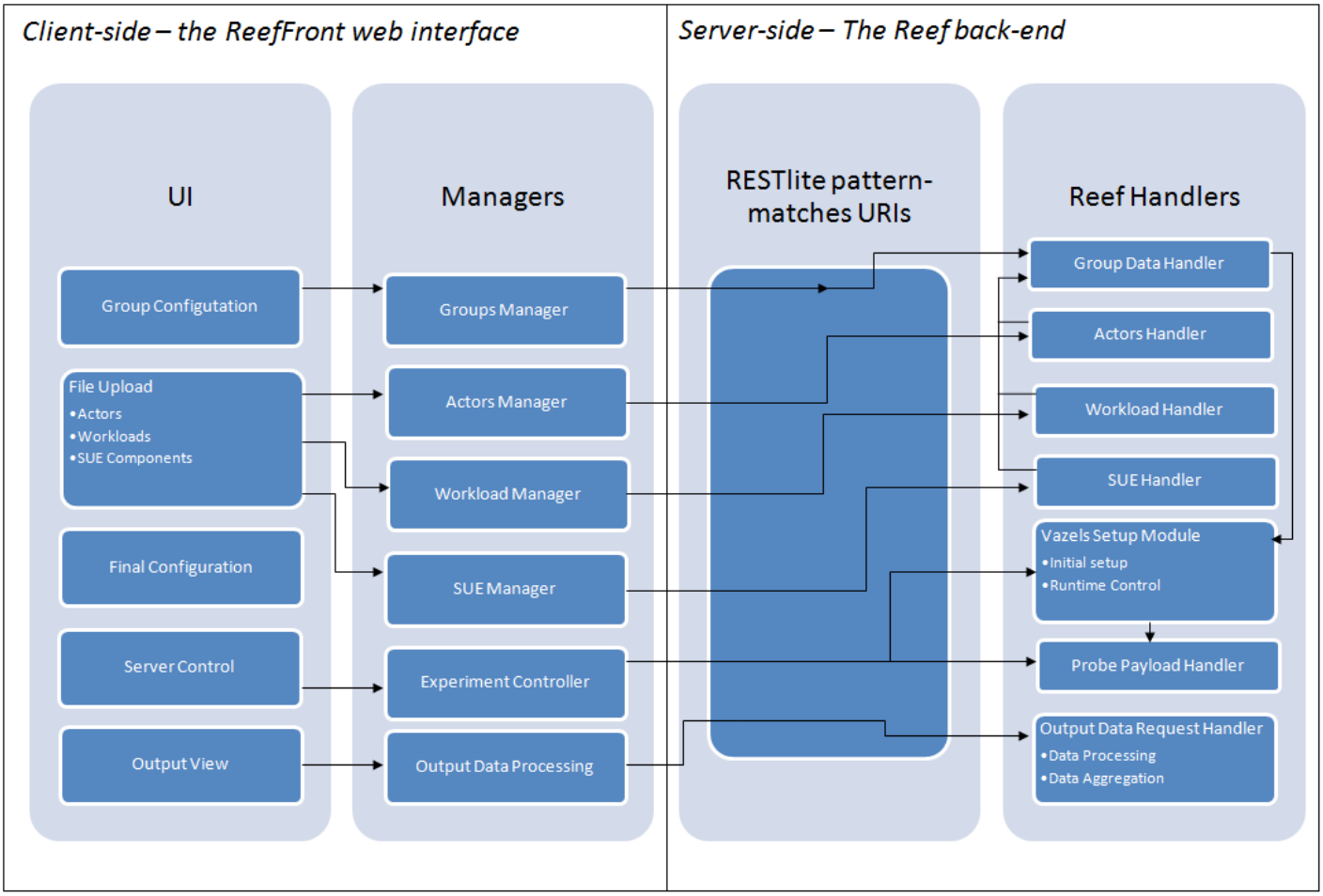

Figure 2: Architecture

Starting the experiment

Under the Vazels system, once an experiment is configured, the user is required to copy a folder from the file system onto the machines which will be taking part in the experiment, and run a file dubbed a "Probe." The Probe is a script that runs configuration on the target machine, and communicates with the Vazels control centre. Once a target machine has a probe run on it, it is assigned to a Group and becomes one of the Hosts in the experiment.

[19]We address this by allowing the user to "go back" - in fact this tears down the server and re-initialises it when they are satisfied with their changes. We felt this should be as transparent as practical to the user.

Whilst this mechanism is portable, it presented us with a problem; it is difficult for us to provide a UI for copying these probes to target machines and running them. We compromised; the user no longer has to locate and copy the probe files from the Vazels control centre, as we provide them in a tar.gz package. However, the user must extract the contents of this archive and run the probes on target machines themselves.

In order to do this, the user has to temporarily leave the web-user interface and run the probes from the command-line. In an ideal world, the user interface would be built such that it provides a consistent and comfortable work environment. Forcing the user to continue using the command-line for part of the setup process seems unintuitive, for a GUI, but providing another mechanism would have been elaborate. Instead, we provide full instructions on how the user can perform this action outside the interface.

Displaying Output Data

Vazels provides all of its output as protocol buffers[20] which are designed to store data in an extensible, recursively-defined way, serialized into compressed files. It also allows the user to specify the data types of fields using a "schema". This allows the application parsing received data to be more aware of what they are handling.

To read the data that has been stored in the protocol buffers, they need to be decompiled using a library available in a number of languages that is built specifically for a particular protocol buffer schema. Our server handler uses a python module to decompile Vazels' protocol buffers and is designed to pass data on to the client in a JSON object which can be read easily by the JavaScript that the client UI is written in. By converting the data into a JSON object we are also able to add our own information, such as group names, which the Vazels system is unaware of in order to make it more user friendly.

The process of displaying the data collected by Vazels to the user begins automatically during the running phase, by sending a request to the server requesting that the data be reloaded. This triggers several activities behind the scenes: firstly, the server issues a command to the Vazels control centre telling it to fetch all the output from the connected hosts. When this has been completed, the server then parses the control centre's output and constructs a JSON object to be sent back to the client. The client must then poll for parsed data, before it can be displayed.

Once the client receives data it rebuilds its current model of the data that is available and organises it in a way that is easily processed by the user interface methods later.

The data is then passed to the interface, which updates a tree on the left of the page to take into account any newly added data, reflecting the makeup of each

[20]Reference: http://code.google.com/p/protobuf/

group (e.g. how many hosts are connected? What data are being recorded?). Different elements of the tree have handlers associated with them that allow the main part of the page to be populated with a graph displaying the data. The reloading process is illustrated in Figure 3.

Currently, reload requests fetch the entire history of the data; an improvement upon this would be for the entire system to be aware of data the client already has and send only incremental updates. This would reduce the amount of network traffic and also the amount of data to be processed, and will prove important if our interface is to scale to the kinds of large experiment for which Vazels is designed.

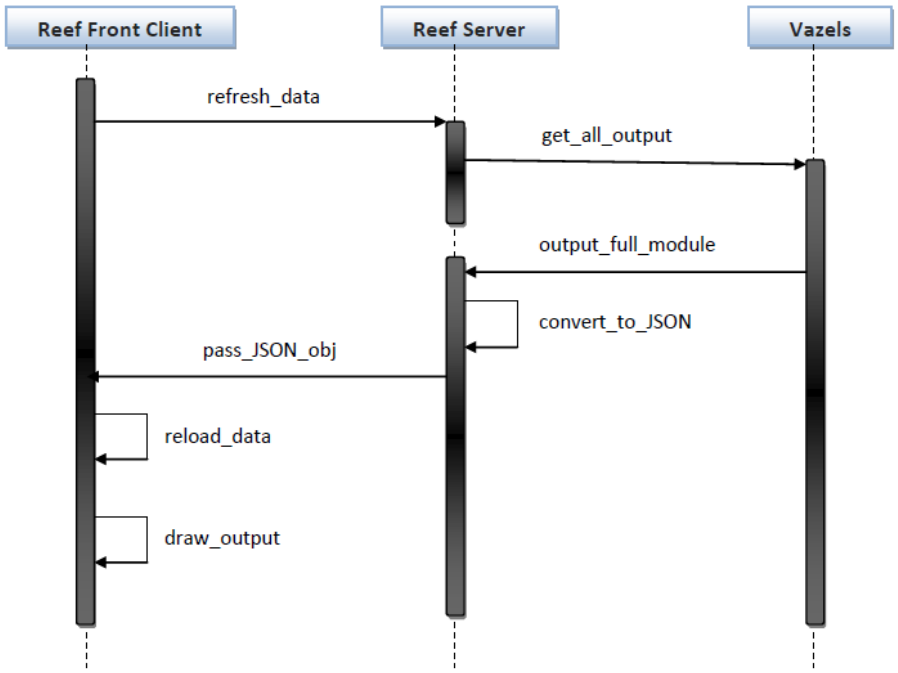

Fig. 3 Output Display Sequence Diagram

Software Engineering Issues

Technical Challenges

Running Vazels

During the project's inception, we began by using Vazels on the command-line. This was a troubled process, and the difficulties experienced help to justify the goal, since it directly addresses some of the issues.

We first attempted to run the system in the Department of Computing (DoC) labs using our own laptops to run experiments over the provided network. We found that newly setup probes were unable to communicate with the Control Centre, but could not easily identify the reason for the problem. It emerged, after some investigation, that whilst our own machines stored their network hostnames locally (and this information was enclosed in the probes that Vazels produced), on the DoC wireless network, they were assigned different hostnames - this meant that probes were unable to locate them.

After working around this problem, (by using lab machines only, or by temporarily renaming our control machine to the name given by the network) we found other minor difficulties, including firewall setup and the Vazels' requirement for an insecure[21] SSH authentication key pair - the private part of which was distributed over the network with the probe.

These further issues should have mostly been addressed by using our server startup script to prepare the system for Vazels.

These difficulties made it challenging to prepare a large amount of test data that could be used to provide the "Running Phase" components with dummy data whilst the other parts of the project were being developed. However it was found that we were able to produce a small amount of data by manually writing the Protocol Buffers that Vazels was expected to produce during an experiment.

Operating System

As Vazels runs only on Unix-based systems, we were required to use a *nix operating system to develop the interface for it. Members of the group used varying operating systems, but the solution was for all members to work on Linux, in order to have a common working platform. This resulted in a loss of productivity for a period of time, as the problems were usually workable and some members were reluctant to migrate from Windows. Eventually all but one of us solely used either Linux or VirtualBox to run a virtual Linux OS on their machine. The other member focused on aspects of the project that did not directly Vazels to run, such as interface design.

[21] Normally when generating an SSH key, one would password-protect it. Vazels requires users to waive this security measure.

GWT

As Google Web Toolkit is relatively new and still evolving fast, it was perhaps inevitable that we would encounter challenging bugs and incomplete documentation. One of the more important problems while developing was that the GWT development mode under Linux can only be used with Firefox. Therefore to test on other browsers, it was necessary to perform a lengthy compile (under development mode, GWT updates its output - our web page - “on the fly” by making use of a browser plug-in that interfaces directly with the IDE). As we were limited to developing on one browser, it was difficult to guarantee support for other browsers. In fact to use some features on Firefox we were forced to use deprecated code as the most recent version did not function correctly. This degrades functionality on some other browsers but can be fixed after future releases of GWT address these issues. This has directly affected one of our core project goals of being “platform agnostic” - support for browsers other than Firefox is limited.

Git

Each group member had a different level of experience with Git prior to the project starting, from complete novices to those comfortable with the majority of features provided. This lead to a steep learning curve for some members of the group before being able to start work on the project itself.

Over the course of the project some issues arose with the use of Git, caused by user error, for instance most users were unaware of the “rebase” command which allows them to update their working branch to include all the latest changes from another. Bug fixes were duplicated, and merge conflicts were generated which were difficult to solve further down the line.

Testing Issues

As mentioned in Project Report 3, the fact that our project was a GUI meant that unit testing was given a low priority until a later stage than would normally be expected in developing a large project.
As we discussed in that report, we were sceptical about spending too much time trying to attain good test coverage of code in the user-facing front-end - the easiest way for us to test this is to start our program up and “see whether it looks right.” Having prepared scripts (“user stories”) to be followed when testing the UI, we had a consistent and method for manual testing.

This was perhaps a short-sighted view, and we did experience behavioural regressions in parts of our projects - however, these problems tended to lie not in functional aspects of the code (e.g. a function returning the wrong value), but in parts that were harder to unit-test (e.g. certain data wasn’t being properly synchronised with the server). Over the course of the project, we had a lecture on Unit testing, however we did not find it applicable to our project; it did not address how to write more testable code, rather what was meant by ideas such as e.g. code coverage.

Towards the end of the project, when it was perhaps too late, we did discover one resource in particular that gave some very practical and usable pointers for designing more testable code: (https://docs.google.com/present/view?id=d449gch_2603sf622cs).

User Validation

From Project Report 3:

> The decision to produce a web-based UI was discussed at the time with our supervisor, and his input was considered during the design of the architecture. Whilst the concepts of AJAX/REST-based UI and running an interface to a service on a remote machine took a little explaining to some members of the group and to the supervisor, we feel that we eventually settled on what was probably the most practical solution.
>
> The design of the UI itself was not the key concern during the development phase of the project. In this phase, the priority was overcoming the difficulties of interacting with a command-line service from our web-server, and of producing the front-end capabilities necessary to properly interact with our web-server. This may seem counter-intuitive, since the main aim of the project was to produce a usable GUI, but it seemed to us that a GUI would be useless if it could not properly control the service it was designed for, and for that reason we focused on the functional aspects of our project until a relatively late stage, when we assigned a group member to produce a visual prototype of how the project might eventually look to the user.
>
> To validate this design, we first considered it as a group, then discussed it with our supervisor, who asked certain questions about the way, for example, lists of machines involved in the experiment would scale. Following this discussion, we refined some of our user stories, and compared them to our design. The aim here was to check that users would be able to easily navigate the interface and achieve the tasks we predicted for them, and we found this method of validation gave us confidence to proceed.

Collaboration challenges

Differing understandings

Despite our best efforts to use methods such as code review and a wiki for knowledge management, members of the group had vastly different understandings of certain parts of the project. Towards the beginning of the work, this was limited to areas such as the way Vazels operated, but as the project grew, there were areas that each member of the group found themselves unfamiliar with. When large sections of the code were refactored to fit with an MVC architecture, group members found that the code they had written was changed by others, and when they returned, progress had to start with re-familiarising themselves with it.

There was also a lack of “over-arching” documentation for many parts of the code. This issue was addressed later, with much of it eventually being added to the Wiki, but throughout the development process, members of the group found themselves misusing classes that had been written by others. While JavaDocs for the individual functions they were calling may have been complete, JavaDocs can be very specific, and are often more helpful to a person already familiar with a piece of code.

Despite the limitations of inline code documentation, we did take the time to prepare scripts that would interface with the JavaDocs and Sphinx tools to prepare and upload documentation to our repository at http://jelford.github.com/reef

Differing workloads

The following is adapted from Project Report 2:

> An issue that arose was that Andy spent longer on the project than most others, and produced much more useful output. It would have been unfair to ensure everybody made this level of commitment, but nonetheless, this led to an uneven, and unfair distribution of work. We did not see it as necessary to provide a maximum time commitment, but wished to avoid any resentment that may arise from a feeling of unbalanced contributions. As such, agreeing on a universal minimum ensured that even the hardest-working members of the group have agreed on an acceptable level of work for the others to be doing.
>
> On a related issue, that of group members having different levels and areas of skill, we have had some issues. Naturally, this has led to some people being more productive, having already been familiar with some of the technologies and concepts involved, e.g. Git, AJAX. We gave the tasks suited to particular members to those people, for example Andy had more input on the server side as he has experience in this. We have tried to ensure that each member of the group recognises the time spent on tasks by other group members. This was helped by a discussion at the meetings not only of what has been achieved but also of the obstacles that have been encountered, which leads to an understanding of why a task may not have been achieved. It was effective in achieving this goal, whilst also maintaining a high level of productivity from the most active members of the group.

Version Control, Issue Tracking, and Knowledge Management

The following is adapted from Report 3:

> We used Git for version control, so that we could all work on separate or overlapping parts of the project without having to deal (immediately) with conflicts. Git’s intelligent and quick merging algorithm meant that solving merge conflicts was relatively painless once all members of the group got used to the tools. To begin with, any large or new operations in Git had to go through one member of the group who had most experience with the system

- which was something of a bottleneck - but now all members of the group are able to deal with the basic functionality. This has been a useful tool in allowing us to work quickly and independently, whilst keeping track of the work we've all been doing.

We originally hosted our project on ProjectLocker[22], which provided us with a hosted Git repository, as well as a Wiki and Trac[23] issue tracking. This solution allowed us to keep our code private while making use of a fully-featured hosting option with a good issue tracker. Using Trac was a good way to see outstanding tasks and also measuring progress. The Wiki provided a good place for knowledge pooling (so that each member of the group could document what they'd done all in one place). However, we had some issues with the host, including high latency, and a period when our files became totally unavailable. Obviously this was a problem for us, so we moved our files to the more reliable Github[24], which similarly provides issue tracking, repository hosting, and a wiki. Moving between hosts mid-way through the project might have been expected to cause problems; however, since Git is designed to work with multiple remote hosts anyway (being a distributed version control system), the transition was smoother than expected. Now that the project is being hosted at an open-source location, the need for good documentation and an up-to-date wiki is more pressing (members of the public, and indeed, potential employers can potentially see our code).

For the duration of the project, the online Wiki did not prove as valuable a tool as we might have hoped. The problem here was perhaps that there was little immediate incentive or reminders for people to go back and update it, unless they were specifically assigned that task. This is a problem that we would each want to address if we decided to use a Wiki for knowledge management in a future project.

Organising meetings

Communicating proved to be difficult at times, with varied timetables and academic commitments. Originally, we intended to have a meeting every 2-3 days; however, realistically we saw this as unnecessary and difficult to plan. However, informal, ad-hoc meetings between varying subsets of the group occurred on most days - this was very achievable, since every day most group members were available on campus. Meeting in person was the standard and also was the most effective means of communication. The frequency of these impromptu meetings led to rapid feedback about progress being made and helped to address difficulties. Alternatives used were telephone and Skype conversations; these were less effective with neither allowing a shared view of the code. However, telephone calls provided a suitable means for short, immediate queries.

[22] http://www.projectlocker.com/

[23] http://trac.edgewall.org/

[24] https://github.com/

We used Doodle[25] to help organise meetings at a time that would fit into the schedules of all team members - the most regular issue being other academic commitments and Simon's job each Wednesday. Doodle helped us to continue having meetings where all 5 members were present, although occasionally this proved impossible. We used Google Calendar to schedule these meetings for everyone, which meant that we didn't have to send out further communications to finalize meeting times (Google Calendar synchronized with our personal calendars).

Code review

Code review played a useful part in the project. Originally we were optimistic, and set a target of two code reviewers for every update to the code. This proved unacceptably time-consuming and greatly increased the delay between code being written, and code being usable to other members of the group - blocking progress on other features. We decided upon a minimum of one code reviewer for all code, which worked well. This code review happened before each branch on the Git version control system was moved into the master branch, noting that each new feature was coded on a new branch. As much as possible, this code reviewer was the member whose work most corresponded to that of the coder, so that the reviewer already had a good idea of the issues involved. For example, if anybody made a change to the server-side code, Andy or James would review this code, since they were responsible for most of the work on this section. However, if errors persisted, a third person would also review any code, and work on fixing the issues.

Summary of Contributions

The following provides a very general summary of the main tasks each member of the group worked on, along with an (approximate) estimate of the number of hours spent working on the project, e.g. time in group meetings not considered.

Andy (259hrs):

- Initial setup of the RESTful server
- Backend handling of Workload and Actors configuration
- Automatic documentation generation (PyDocs and JavaDocs)
- Client side low-level server interaction
- Refactoring
- Re-organising client-side architecture
- Preparing test data with Vazels

Andreea (89hrs):

- Uploading of groups
- Client UI design

[25] http://www.doodle.com/

- Graphical styling
- Project reports

Chris (77hrs):

- Uploading of actors and workloads
- Work on the “Final Configuration” stage - assigning Actors to Workloads and Workloads to Groups
- Project Reports

James (156hrs):

- Backend handling of Groups configuration
- Preparing test data with Vazels
- Administrative tasks
- Project Reports
- Finalizing Setup-phase UI

Simon (97hrs):

- Running phase UI
- Experiment data collection and interpretation
- Experiment representation
- Project Report

See Appendix 1 for a table from Report 3, containing a more detailed breakdown.

Validation & Conclusions

Project Success

The final deliverable of the project provides an interface through which a user can perform all tasks involved in setting up an experiment for running on the Vazels system

Project Aims - The Deliverable

To address the success of the project, it is worth returning to the Key Requirements section of Project Report 1 (Project Inception) and we will discuss in turn the extent to which each requirement has been met:

Setup Phase

In this stage the user will configure settings for Groups, the number of hosts in them, and which actors and workloads will be loaded onto the hosts in a given group, as well as where the SUE will be situated. The user must be able to:

Set up Security

> *Partially Complete* - the user is able to set up password restrictions on using the server, and is not required to configure their SSH keys - however, they must be certain that the machine on which the Vazels control centre is to run is able to recognise its own SSH key as a known host (this is a limitation of Vazels).

Specify crucial constructs of the test

> *Complete* - The user is able to specify Actors, Groups, SUE Components and Workloads successfully, which are the vital things that an experiment needs before it can be run.

Upload the required files

> *Complete* - The user is able upload any files required for an actor or SUE component to execute. They can also upload a pre-prepared workload file.

Running Phase

In this stage the user will be able to monitor their running experiment. They must be able to:

Display which hosts are online or offline at any given time

> *Partially complete* - At the time of writing, this functionality has not been fully tested and has been omitted from the user interface.

Have the Control Centre collect data from attached hosts

Completed - it is possible to send a series of commands that will cause the Vazels control centre to gather all output data from a running experiment.

Shut down the server

Partially complete - a running experiment can be finalised and raw output data can be collected, however the server will not close (it must be terminated on the command-line); instead it returns to the "Setup Phase," ready to run a new experiment.

Other Key Requirements

The user must be able to administer a Vazels Control Centre running on a remote machine

Completed - Once the server is running, it can be connected to via any machine with access to the server through port 8000.

The interface must run on at least Firefox and Chromium

Partially complete - As discussed in the section above on Technical Difficulties, users are limited in their choice of browser. Chromium and Midori browsers will not correctly render the page, while Firefox and Opera support all tested features. Since 30% of web users already use Firefox[26], and it has a stable release for every major platform, we find this an acceptable compromise.

The back-end for the Web interface must be easily configurable by an inexperienced user

Complete - We provide a simple script that handles the setup of the server, and configures the file system and security based on input from the user.

There must be reasonable security restrictions.[27]

Complete - Access to the user interface requires basic HTTP authentication using a password the user specifies during the start up of the server.

Extensions

Provide the user with a more interesting display of the data

We have started to expand the way experiment data is displayed to the user. We surmised that the most important data to visualise first would be a series of numerical values that Vazels actors can store at any point during the

[26]According to the W3 Counter at http://www.w3counter.com/globalstats.php?year=2010&month=12

[27]Unauthorised users must not be able to gain access to the Vazels system via Reef.

experiment since this type of data would be the hardest for an experimenter to understand in a raw form and was also the simplest for us to present data to the user in a meaningful way. We display a line chart containing a particular variable's values for each host within a group, against time. Any hosts that do not have data for that particular variable are not displayed. To further this work, we would like to be able to make the system more aware of the data that it produces a chart for, e.g. if there is only one value per host then it might be better to draw this as a bar chart, and this presentation is inappropriate for string output.

Prefabricated tests

This feature has not been implemented, although it is possible that a library of actors could be provided to users at a later date for them to include in their own workloads.

Visual "point and click" workload building and group configuration

This feature was deemed too large a task to consider starting.

Saving

Provided the user has not deleted their working directory it is possible to stop the server and restart it later and maintain all the experiment configuration data that the user has entered. Any experiment data that has been produced is also stored for use later. The user also names their experiment at startup, so it is possible to return to an old project after starting a new one.

Usability for users with no prior experience with Vazels

Obviously, a GUI makes progress towards this; however, Reef still requires an understanding of the concepts and terminology that Vazels uses. Thus, we have not expanded the potential user base of Vazels as much as we might have hoped for. Within the interface we do provide a link to the project's wiki with a section, albeit limited, created solely for users of the software.

Project Aims - Ongoing Potential

In undertaking this project, we hoped to develop a genuinely useful tool for users of the Vazels system. Given our time constraints, we found it unrealistic to aim to complete all the features we wanted and were potentially within scope for the term. For this reason, and also because we thought that the methods would aid members in contributing independently, we tried to produce the documentation necessary not only for a user of our tool, but also for another developer or group of developers to further improve it.

With this in mind, we took time to produce thorough JavaDocs and PyDocs throughout our code, and also to write more over-arching material on the architecture of the project, and the design decisions taken. All the code and

documentation are available to the general public, hosted on Github (https://github.com/jelford/reef). It is our hope to possibly take this project forward in the future ourselves, or that other developers will be able to contribute - GitHub's fork function makes this task effortless.

Personal Success

A major personal gain that every group member is able to take away from this project is the knowledge of at least one new toolkit or technology. This project has also reminded all group members of the need to manage their time correctly and address concerns in a timely fashion, e.g. by raising a problem with an impending deadline with another group member who may be able to take on some of their workload, or help them to finish it themselves. These skills alone make the project worthwhile from an individual perspective.

GWT was new to all members of the group but it had certain similarities to the Android toolkit, which two of the group members have used extensively, and has given other group members more confidence in considering exploring this other new technology. Learning this new toolkit should go on to help each member of the group both in their personal and professional future projects - for all the technical reasons described earlier, GWT is a powerful toolkit with striking potential for building web applications elsewhere.

An important gain that each member has made is the increased understanding of Git, and its related tools (e.g. Meld used during the merging of branches). Whilst many of us were already aware of version controlling, most had not been the authors of repositories before and so were able to get a better understanding of managing a repository through actions such as branching, rebasing and merging. As we progressed, even the most sceptical of group members came to value the contribution of Git, and all would now be keen to use it in future projects, academically or professionally. Familiarity with Git and version control has inspired group members to relocate some of their own personal endeavours to open source project hosts such as GitHub and Gitorious.

Maybe the key lesson that this project has reinforced is the necessity for good design from the beginning of development. This would have reduced the amount of refactoring that was required by helping group members to see how their classes would fit in amongst the system as a whole, meaning that code review was less drawn out, the code base was more consistent, and there would be less need to re-work old code.

With regards to teamwork, we have had to practice our ability to give and receive criticism in a constructive and professional way. Open and honest discussion within the group achieved better results in the long term, by allowing us to focus on work instead of personality conflicts.

This touches on the point that different group members responded differently to different management styles, with some members of the group thriving in a high-

pressure deadline-driven environment, whilst others required constant gentle nudging - and some required almost no management at all.

Future Success

The majority of our organisation style developed during the project, e.g. at the beginning we were optimistic with regard to frequency of meetings, very short iteration cycles and division of tasks. For this project we simply moved back iteration cycles, and mixed the initial iteration cycles, however, we should have taken a step back (despite the short-term time pressures) to ensure we obtained a more realistic schedule.

At the beginning of the project we were offered the opportunity to discuss with Angel Dzhigarov about the underlying Vazels system that we would be building onto. We decided as a group that we had no specific questions for him with regard to building the user interface. Potentially, Dzhigarov may have had undocumented hooks within his system ready for a user interface to be attached. Therefore some of the intermediate work that we produced may have been done already. Additionally, Dzhigarov may have thought about alternative approaches to developing a web interface, having stated a web interface as one of the extensions that he wants to do on the project's web site. At any rate, discussion would have sped up initial progress, although independently learning about the tools used and options available was valuable. Another benefit that could have been gained from greater communication with Dzhigarov before building upon Vazels would be gaining his interest and enthusiasm. This is of some importance since we want the project to continue after our end, and would like it to become the fully functional web interface which Dzhigarov seeks. This would lead to the product being used, which is a goal that all members see as very valuable and worthwhile.

How the project can be extended:

We have already provided a series of extensions which we had hoped to achieve if we had finished the other aims earlier. All those aims which are incomplete would be a valid extension. To implement all functionality of Vazels would be the primary, recommended extension; until more of the functionality is implemented, it is unlikely to be a viable tool. However, we believe the open source code, which, it is worth noting, has been refactored many times solely for the purpose of clarity, provides a good basis on which to build and extend in order to provide the necessary functionality.

Appendix 1

Breakdown of contribution (extended from the same section in Report 3):

Week Ending 25/10

- **Andy** 2hrs: Learning how Vazels works, 18hrs: Debugging Vazels - initial problems involving infrastructure in Department of Computing's lab facilities.
- **Andreea** 3 hrs: Learning about Vazels. 1hr: Learning about similar tools on PlanetLab.
- **Chris** 3hrs: Learning about Vazels. 2hrs: Looking at GWT and Git.
- **James** 1-2hrs: Looking for toolkits, (GWT, Git, svn). 1-2hrs: Going through GWT tutorials. 10hrs: Working with Vazels & trying to run test experiments. 3hrs: Finding a hosting solution for Git and investigating issue tracking systems. Getting those set up. 2hrs: Formalising initial project aims and writing to-do lists from discussions in emails. 2hrs: Going through Output Parser code with Simon.
- **Simon** 4hrs: Learning to use Vazels, running experiments and understanding how outputs are formatted. 4hrs: Writing test code that could parse the Protocol Buffers provided by Vazels. 4hrs: Writing a python module that could eventually be called to provide the output as a JSON object. 3hr Following git manuals and getting written code put into the repository. 2hrs: Going through code with James.

Week Ending 01/11

- **Andy** 1-2hrs: Researching toolkits (RESTLite, GWT), 35hrs: Writing a basic server and client, including a persistent configuration manager and authentication server side. Writing a configuration editor, local cache, and AJAX request helpers client side. Also writing a proxy server for GWT to forward requests to our real server during testing (GWT has a "testing" phase which complicates SOP restrictions on modern web browsers).
- **Andreea** 5hrs: Researching toolkit (GWT). 1h: Researching Git (trying a tutorial) 3hrs: Investigating transactions (a Vazels concept) and their use for our project. 3hrs: Project Report 1.
- **Chris** 5hrs: Researching GWT, and installing compatible Eclipse plug-in. 3hrs: Learning about HTTP Post requests, and the necessary server interaction. 3hrs Project Report One. 1hr: code review.
- **James** 13hrs: Writing Project Report 1. Less than one hour: Putting administrative documents into the Git repository. 1hrs: Code review. 2hrs: Enabling starting and stopping of Vazels with basic error checking (client and server-side).
- **Simon** 3hrs Writing the handler for the rest server to connect with the output parsing python module.

Week Ending 08/11

- **Andy** 4hrs: Solving toolkit version issue (I downloaded the toolkit three days prior to everyone else and as a result had a different version). 3hrs: Moving settings manager (client) to use a graphical UI builder. 4hrs: With James, working out who destroyed the repository and fixing it. 9hrs: Getting our testing proxy to handle multi-part form data. 3hrs: Code review.
- **Andreea** 4hrs: Dealing with toolkit version issues (incompatible with Andy's). 2hrs: Further familiarising with Git. 8hrs: Coding the Allocate Groups Widget (Groups are a Vazels concept). 2hrs: Learning how to use the UI Binder (building interfaces in XML instead of procedurally).
- **Chris** 4hrs: Creating a GUI which uploads workloads, using GWT. 1hr: Setting up and familiarising self with Git.
- **James** 2hrs: Moving documentation into the Wiki. 4hrs: Running Vazels experiment projects and capturing output for testing by Simon. 4hrs: Merging branches, fixing merge conflicts, fixing issues with the repository. 5hrs: Designing the server-side interface to groups administration.
- **Simon** 2hrs: Taking others' Vazels experiment data and validating that the output module was working. 1-3hrs: Began to think through how it would be possible to display data to the user.

Week Ending 15/11

- **Andy** 1hr: Documentation. 4hrs: Migration of UI components to XML-based specification (a major feature of GWT). 7hrs: Various minor fixes. 5hrs: Modularising server components. 5hrs: Created SSH key generation code, and got the server to save configuration automatically. 2hrs: Refactoring. 1hrs: Project Report 2.
- **Andreea** 2hrs: Coding the client side of the Mapping Restrictions
- **Chris** 6hrs: Allow obtaining of list of workloads previously uploaded to the server. 3hrs Project Report Two
- **James** 8-9hrs: Project Report Two. 1hrs: Merging & Code review. 4hrs: updating server-side Groups management.
- **Simon** 6hrs: Experimenting with storing output data client-side that could be updated incrementally as data was received - eventually decided against. 4hrs: Project report 2. 1hr: Code Review.

Week Ending 22/11

- **Andy** 4hrs: Refactoring & Documentation on the Wiki.
- **Andreea** 2hrs: reading on good practice on building GUIs. 2hrs: Sketching the design of GUI on paper. 8hrs: Creating first visual draft of GUI.
- **Chris** 5hrs: Working on uploading workloads to groups - including creating singleton Workloads class to have a local store of data. 2hrs: GUI to upload actors.

- **James** 3hrs: Refactoring, documentation & trivial code changes. Less than 1 hour: Redesign of global configuration management client-side. 5hrs: Updating server-side control of Vazels to monitor output and do more robust error checking/handling. 1hrs: Adding ability for the user to upload the System Under Test (a Vazels concept). 3hrs: Adding ability to properly start the server with all files extracted to appropriate directory.
- **Simon** 6hrs: Experimenting with new ways to display/layout each GWT widget on the page. Implemented a tab layout.

Week Ending 29/11

- **Andy** 5hrs: Fixed a number of unnoticed regressions to do with deployment (not caught by tests due to the GWT test harness functioning slightly differently to the deployed product). 2hrs: Refactoring & improving authentication methods server-side.
- **Andreea** 4hrs Creating the Main Setup Menu Tab (restructuring the UI to fit with the new design)
- **Chris** 2hr: Creating a Workload manager in same style as the (since recreated) Group Manager, 1hr: Code review.
- **James** 1hrs: Trivial code changes and refactoring. 1hrs: Code review.
- **Simon** 1hr: Code review.

Week Ending 6/12

- **Andy** 12hrs: re-implementing client-side global storage for configuration data. 2hrs: Rewriting code to make better use of existing Python libraries rather than duplicating functionality. 8hrs: Refactoring the code I wrote this week.
- **Chris** 4hrs: Interfacing with new Group Manager to easier upload to server
- **James** 2hrs: Code review.
- **Simon** 4hrs: Started Output Data widget.

Week Ending 13/12

- **Andreea** 13hrs: Applying visual styling.
- **Andy** 20hrs: Code review.
- **Chris** 4hrs: Uploading of workloads to groups - finish. 3hrs: Project Report 3.
- **James** 12hrs: Project Report 3 (Validation). 5hrs: Updating server code to query running experiment for results, as well as monitor status of running hosts. 1hr: Refactoring.
- **Simon** 3hrs: Project Report 3. 5hrs: Building Output Data widget.

Week Ending 20/12

- **Andy** 30hrs: Code review of the interface. 10hrs: Documentation

Week Ending 27/12

- **Andy** 35hrs: Documentation, including the very beginning of Christmas day.

Week Ending 03/01

- **Andy** 17hrs: Code review/refactoring. 4hrs: Bug fixing.
- **Chris** 5hrs: Finish assigning of actors to workloads and bug fixing
- **James** 10hrs: Adding client-side code to allow starting the experiment and monitoring state. 3hrs: Admin & starting Final Project Report.
- **Simon** 5hrs: Improvement work on output display, making graphs neater.

Week Ending 10/01

- **Andreea** 33hrs: Final report
- **Andy** 20hrs Code review and fixes. 2hrs Final report.
- **James** 45hrs: Finalising Workload & Actor assignment widget, adding SUE assignment, Final Report, final bug fixes.
- **Simon** 14hrs: Finalising displaying output data, adding panel to display group data and functionality to download raw data. 24h: Final Report
- **Chris** 22hrs: Final report

Appendix 2

Simplified Class Diagrams

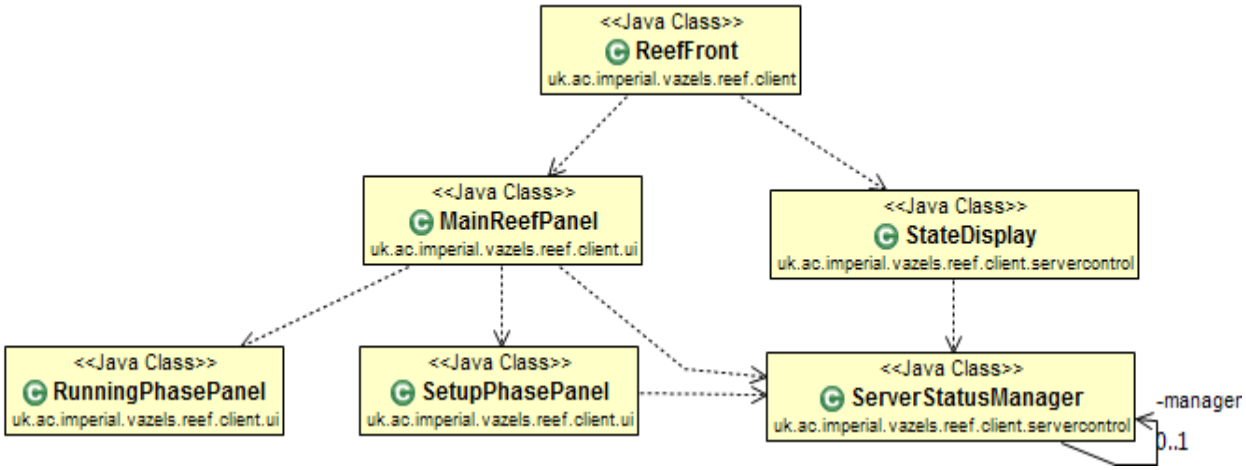

Fig. 1 Main ReefFront Class Structure

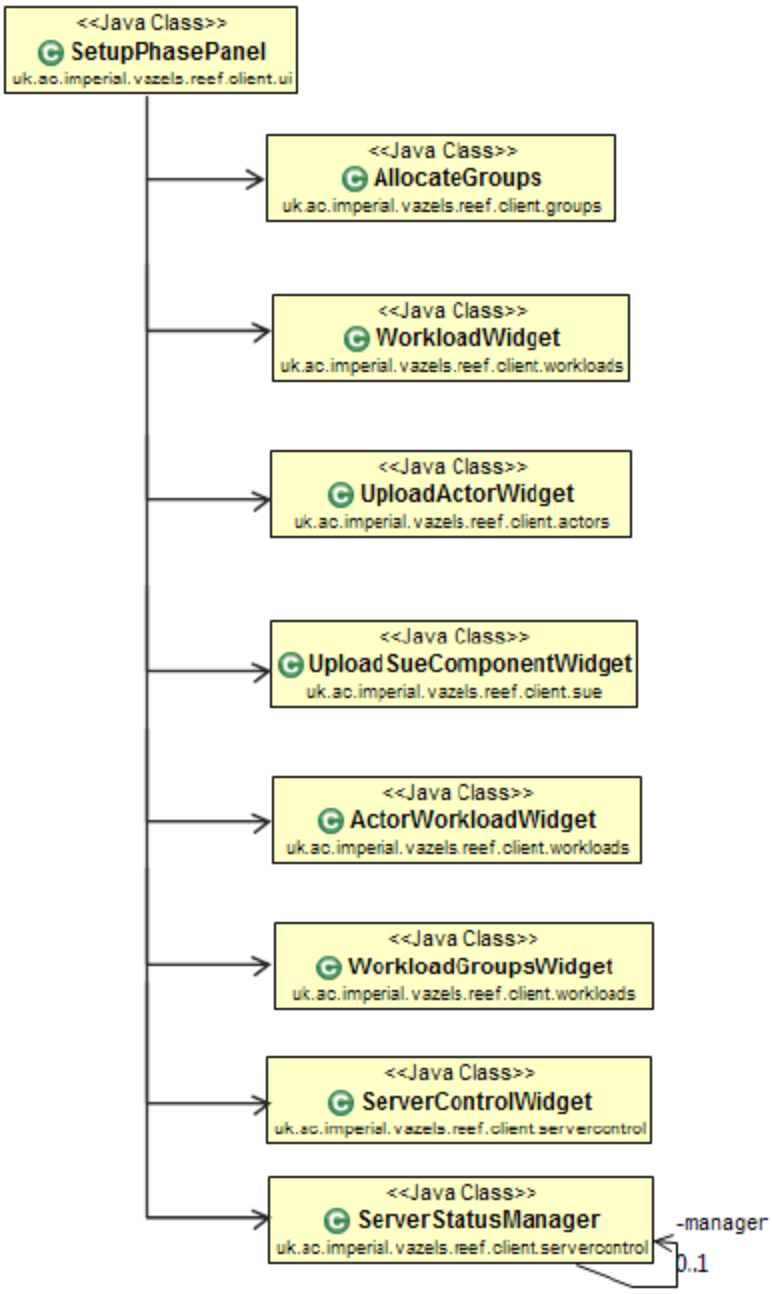

Fig. 2 Setup Phase Class Structure

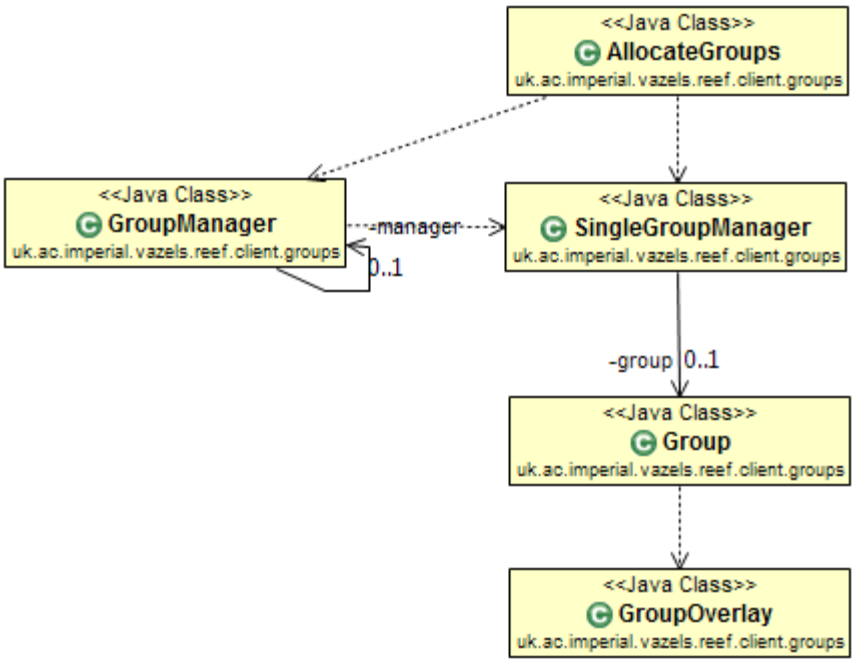

Fig. 3 Allocate Groups Widget Class Structure

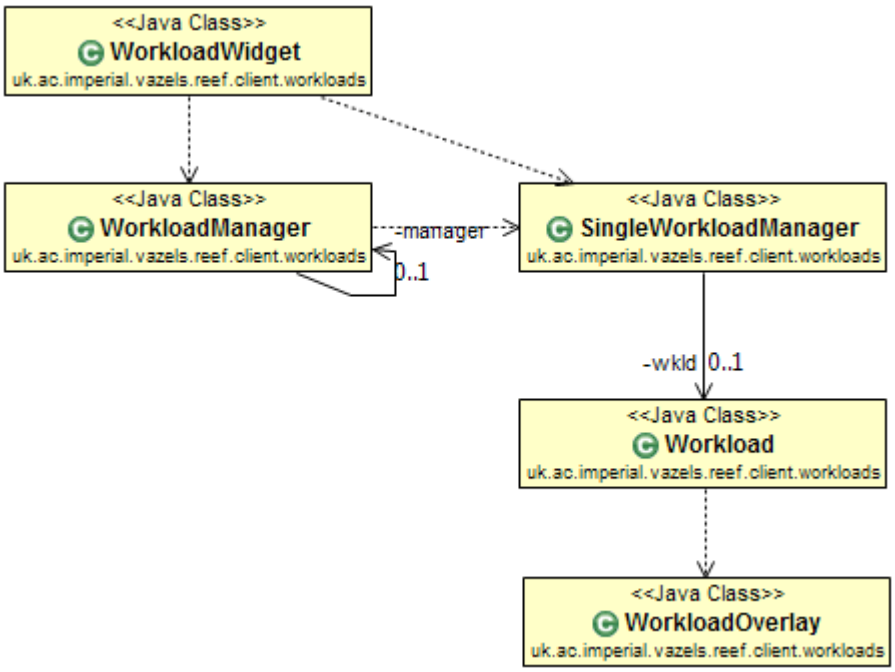

Fig. 4 Workload Widget Class Structure

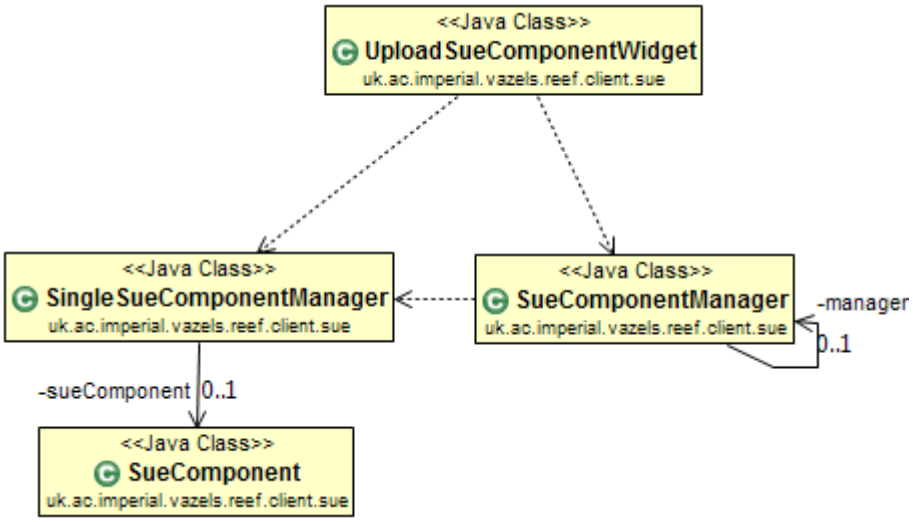

Fig 5. Upload SUE Component Widget Class Structure

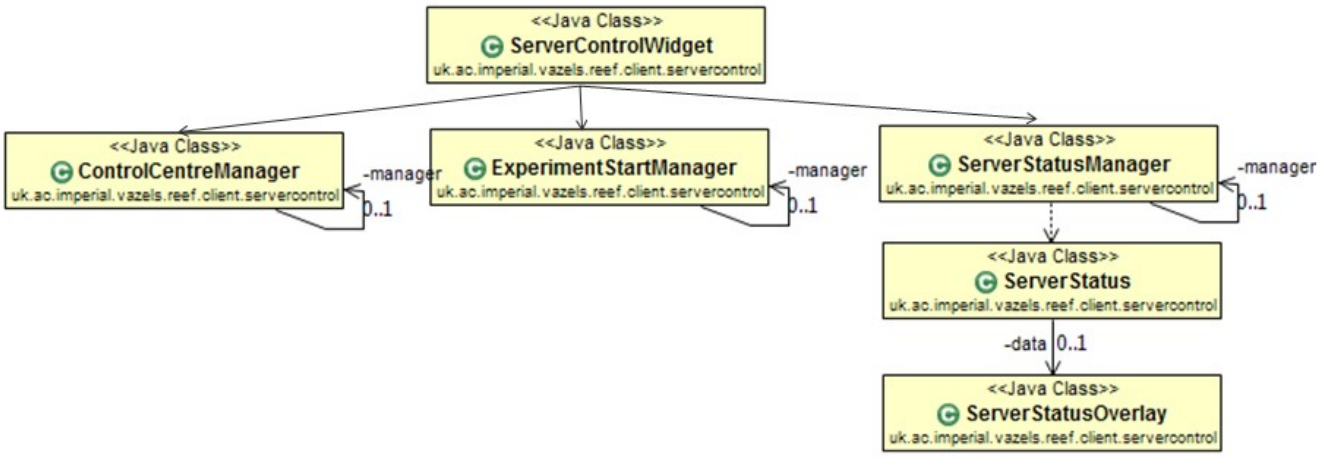

Fig. 6 Server Control Widget Class Structure

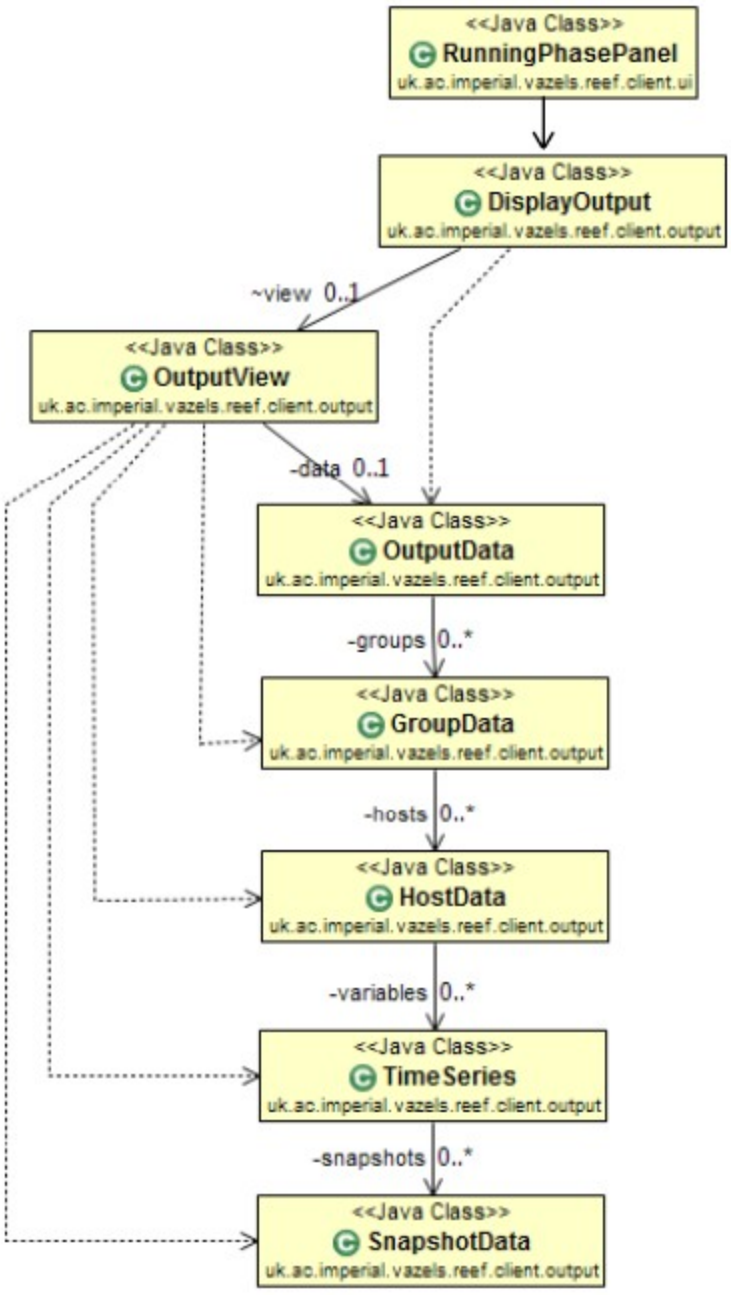

Fig. 7 Running Phase Widget Class Structure