UNIVERSITY OF GOTHENBURG

Testing a distributed Wiki web application with QuickCheck

Master of Science Thesis in Computer Science.

RAMÓN LASTRES GUERRERO

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2012

Testing a distributed Wiki web application with QuickCheck.

Ramón Lastres Guerrero

Examiner: John Hughes

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2012

University of Gothenburg

Master Thesis

Testing a distributed Wiki web application with Quickcheck

Author:
Ramón Lastres Guerrero

Supervisor:
Dr. Thomas Arts

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science in Computer Science

in the

Department of Computer Science and Engineering
University of Gothenburg - Chalmers University of Technology

May 2012

UNIVERSITY OF GOTHENBURG

Abstract

Department of Computer Science and Engeneering
University of Gothenburg - Chalmers University of Technology

Master of Science in Computer Science

Testing a distributed Wiki web application with Quickcheck

Ramón Lastres Guerrero

Web applications are complex, heterogeneous and dynamic. To ensure their correct functional behaviour is a difficult task and by simply using unit testing and manually generated test cases it may be difficult to detect errors present on them. We consider that the use of Quickcheck, a testing tool that provides automatic test case generation and state-based testing, can be a good way to test web applications. To show that, we have tested a web application that uses a distributed database system by creating a model of it. By using this Quickcheck model we have generated and executed test sequences that address the dynamic aspects of the web application. Using this method we were able to find an error that lead to inconsistencies in the data structure of the application.

Acknowledgements

I want to thank my project supervisor, Thomas Arts, for his guidance, help and support during the development of my project. Despite his very busy agenda, he could always find time to help me to advance with my work. He has been a great mentor.

Thanks to Laura Castro for giving me this opportunity and for being always nice and helpful, and also a great teacher.

Thanks to John Hughes for having provided valuable feedback and accepting to be my examiner.

Finally, I also want to thank the *Swedish-Spanish Foundation* for helping to support my studies in Sweden.

Contents

List of Figures

Chapter 1

Introduction

Nowadays many web applications, having a great number of users, handle redundant data storage together with a big load of concurrent accesses and data changes. This makes scalability a key issue. Usually, in a highly scalable system there is not a single point of failure, requiring for the back-end database storage system used by this kind of web applications to be distributed. Ensuring data consistency and that the business rules (logic and constraints at the user level) hold at any time are two important issues. This is a difficult problem when the web application uses a distributed database as its back-end storage system, something that is growing in popularity. In such case, to guarantee data consistency is more complicated than usually. This is because, as the *CAP theorem* [1] states, it is impossible to have a distributed database system that ensures consistency, availability and partition tolerance at the same time. Distributed database systems usually sacrifice consistency in order to get scalability and faster responses, becoming eventually consistent.

Web applications are also complex and heterogeneous. They are *dynamic* and usually have several components created using different technologies that interact between them. This makes difficult to ensure its correct functional behavior. Many potential errors present in them are difficult to detect by simply using *unit* testing and manually generated test cases. Therefore there is a need for testing methods that can target complex use cases and specifically the dynamic aspects of the web applications [2]. Considering this fact, we think that the use of a testing tool that provides a way to model *stateful* systems and automatic test generation and execution can provide a good way to address this issues related with functional testing of web applications. With the purpose of knowing whether this is right or not, we have tested a distributed web application using *QuickCheck* [3] [4], a testing tool that provides random test generation and state-based testing among many other features. Using it, we can address the *stateful* behavior of any dynamic web application. Also, to ensure the correctness of this kind of web applications where data consistency is a key issue, it is necessary to perform concurrent writes and accesses to the data when testing. This is because consistency issues that should appear when several users access the application at the same time may be impossible to detect if the testing is limited to sequential requests. In this thesis we show that QuickCheck can be

used to simulate many users accessing the web application in a highly concurrent way and to check that the business rules hold at all times.

The web application under test, called *Wriaki* [5], follows the principles of *REST* [6]. It is a *Wiki* application and provides basic functions common to every Wiki. It is designed to be able to handle lots of users and is highly scalable. It uses a distributed database called *Riak* [7], which, as mentioned before, provides scalability and partition tolerance, and is able to support high concurrency, but cannot guarantee data consistency. In such a *Wiki* application, many users editing an article at the same time while others are accessing it, either in its current version or an old one in the history, is a normal thing. Considering this and the distributed storage system under use, to guarantee data consistency is both an important and difficult issue. In section 2 we describe *Wriaki*, *Riak* and *Quickcheck* in detail.

Our approach, which is described in section 3, consists on modeling the different web browsers which perform the requests using *Quickcheck's* state machines to test the web application at the level of the HTTP protocol and for that an understanding of the format of the HTTP messages is necessary. By using this method we have developed a model that provides automatic generation and execution of test cases that cover most of the functionalities provided by the web application and that addresses its dynamic behavior. It can also check its correct performance towards concurrent requests by executing several of the test cases in parallel. As a result of this, we have been able to identify one error in the web application that lead to inconsistencies in the data structure used to store the information in *Riak*. Details of the implementation and the error found are provided in section 4.

Finally, in section 5 we present the related literature covering functional testing of web applications, in which we found little coverage of the dynamic aspects of them; and we present the conclusions in section 6.

Chapter 2

Background

2.1 Riak

Riak [7] is an open source distributed database system developed by Basho [8] that provides *key-value* storage. It is influenced by the *CAP theorem* [1], which states that only two of the three properties, *consistency* (C) *availability* (A) and *partition tolerance* (P) can be guaranteed at the same time. Most traditional database systems have chosen to support C-A, by trying to minimize the risk of P by running on one computer or a very reliable network, normally connecting computers that are placed in the same room. For non-distributed databases, C and A are guaranteed, since P is not an issue. It becomes an issue when the database is distributed. The reason to distribute is to increase performance in presence of millions of users, therefore A is a must. The choice is offering either P or C. Offering P allows the database to be more reliable, which is extremely important since having several nodes at the same time increases the risk of losing data, so to guarantee no possible data loss is worth it to weaken the consistency demands. Thus Riak is designed around the principles of *availability* and *partition tolerance* rather than *consistency*, which makes it possible to have several versions of an object at the same time, having data conflicts that the user needs to handle.

A Riak *cluster* is formed by *nodes* connected to each other that communicate and share information between them, organized in a ring structure. Nodes can be added or removed dynamically, and the system redistributes the data accordingly and manages data replication automatically. Nodes can also be configured to use different storage systems as their *back-end*. It includes elements like *buckets* or *links* to organize the data, providing a way of introducing some logic on it, unlike simple key-value storage systems. Buckets are independent key-value spaces or areas within a Riak cluster, and links allow objects stored on a Riak bucket to point to other objects stored in any other bucket.

Developed mainly in *Erlang*, Riak is a distributed system, and makes use of *MapReduce* [9] to perform querys and fetch data. MapReduce is a programming paradigm that allows the execution of highly parallel computations over big sets of data in a distributed or parallel way.

Every MapReduce computation consists of a *Map* phase, where data is fetched in parallel from different nodes or workers, and a *Reduce* phase, where the fetched data is used in some way to produce an output. Riak takes advantage of this paradigm, and considering that data is stored in different nodes around the cluster the *map* phase is performed locally in every node, which sends the output to the node that started the MapReduce operation and performs the *reduce* phase. In this way, we can say that instead of sending the data to the computation, the computation is sent to the data, minimizing the quantity of information exchanged by the nodes. The user can specify an input for the query (buckets and keys) and a *reduce* function written in *Javascript* or *Erlang*.

For the communication with the clients, Riak provides both, an HTTP API and a *Protocol Buffers* [10] based API, and through them one can perform any kind of operation on the Riak database through any of the nodes that form the cluster. Clients are available for most popular programming languages.

2.1.1 Handling consistency issues

Considering the distributed nature or Riak, where any node can handle a request, and not all nodes need to take part on every request, it becomes necessary to tag every object version to be able to know their order in an eventual consistency problem. For this purpose Riak uses *Vector Clocks* [11], which is an algorithm aimed to provide partial ordering on distributed systems. When an object is created, it is tagged with an initial vector clock, which will be extended in the following versions. Having two objects with the same key and two different vector clocks, Riak will be able to know if they are descendant of each other and which one is older, if they have a common parent or if they are not related at all. This will allow the user or the application itself to handle consistency problems. When performing a write, the client always needs to retrieve the current vector clock before, and then request the write using that vector clock. Any write that is performed using a vector clock that is not descendant of the current one is considered concurrent. This can be either, for example, two or more writes using the same vector clock, or one or more writes performed with an old or stale vector clock.

Every bucket in Riak can be configured to either allow multiple writes with just a simple change in its configuration. If they do not allow multiple writes, having a concurrent write situation will simply result in an error. If concurrent writes are allowed, in a situation of concurrent write, *siblings* will be created. This is just an object that contains all the multiple versions resulting from the concurrent write. Having this situation, the conflict needs to be solved, either by presenting the versions to the application final user to let him decide the current version, or automatically applying specific rules on the application level. In fig. 2.1 we can see an example of a situation that leads to the creation of three siblings in a bucket where multiple writes are allowed. We can observe that the two first editions are performed in a right way and they don't lead to any conflict, but when the *User1* does the third write, the *User2* retrieves the vector clock before the write is performed, and thus obtains and uses the same vector clock for the

write, leading to the creation of two siblings. Finally, *User1* performs a write with an old or stale vector clock, which adds a new element to the two previous siblings.

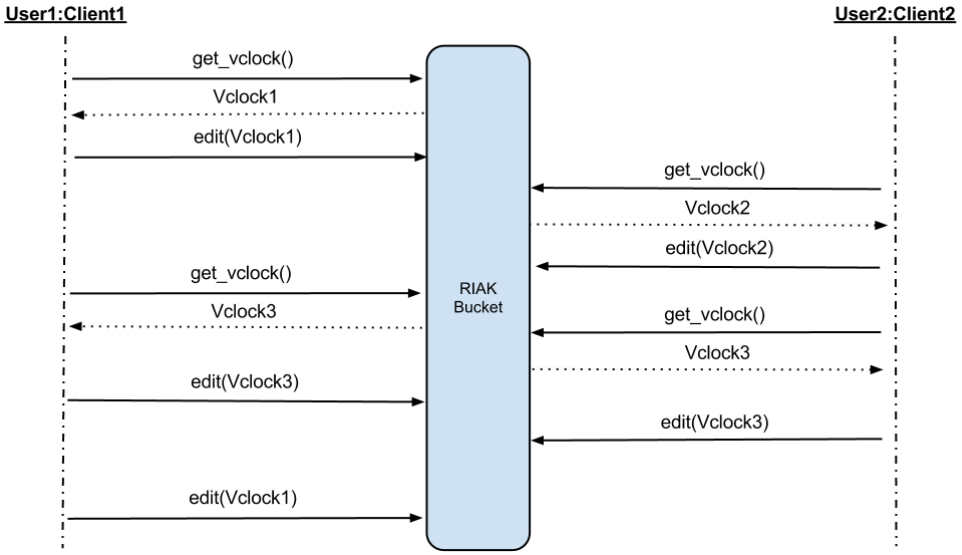

FIGURE 2.1: Possible sequence of editions leading to siblings.

2.2 Webmachine

Webmachine [12] is an *Erlang REST framework* developed by Basho . *REST*, which stands for Representational State Transfer, is a nowadays very popular architectural style based on several principles and design choices that relate and can be applied when building web applications or services. It was first introduced as part of a PhD thesis dissertation [6]. In practice, *REST* based web applications, usually known as *RESTful*, follow some important design principles [13]:

- Use of the HTTP methods strictly following the protocol documentation, which means using the method *GET* strictly for retrieving the resources, *DELETE* to remove them and so on.
- They are stateless, HTTP requests always include all the necessary information to handle them. This simplifies the server implementation and improves scalability.
- The resources or sources of information available are referenced by a unique identifier which is the *URI*, and it should be self descriptive in relation to the resource it points to.
- They always use a standard format to represent and transfer the information or resources, like *XML* or *JSON* (Javascript Object Notation).

Webmachine follows the principles of *REST* and is intended to simplify the development of well behaved web applications, by handling the HTTP protocol semantics and allowing the

developers to define the relevant web resources and focus on the application behavior in itself. It makes use of the Mochiweb library, and it has been used to build Riak's HTTP interface, which is used by the different available clients to communicate with Riak.

2.3 Wriaki

Wriaki [5] is a web based *Wiki* application developed by Basho as an example to both, illustrate different strategies to store data using Riak, and how to build web services using Webmachine. It provides a basic user interface that allows users to sign up for editing, read and edit articles and to see a history of different versions for each article. It also provides an option to see differences between two different article versions and the possibility of searching through the text of articles stored, which makes use of Riak Search and only works if it is installed on the Riak version under use. In fig. 2.2 we can see the view of a possible article stored in the Wiki application. A user (User1) is logged into the application and the links provide a view of the article (view), a form that we can use to edit it (edit) and a history list where we can see all the previous versions listed in chronological order (history).

FIGURE 2.2: Possible view for an article in Wriaki.

Through the edit link we can access the edit view of the same example article, as shown in fig. 2.3. Using the form the users can write the article using *Creole 1.0* [14], a common *Wiki* markup language used across many different Wikis. This allows users to easily give format to the articles. When editing an article, the user needs to write a revision message that describes the changes. Finally, the web application also provides some other functions that allow registration of users, to change the user information, see differences in articles, etc.

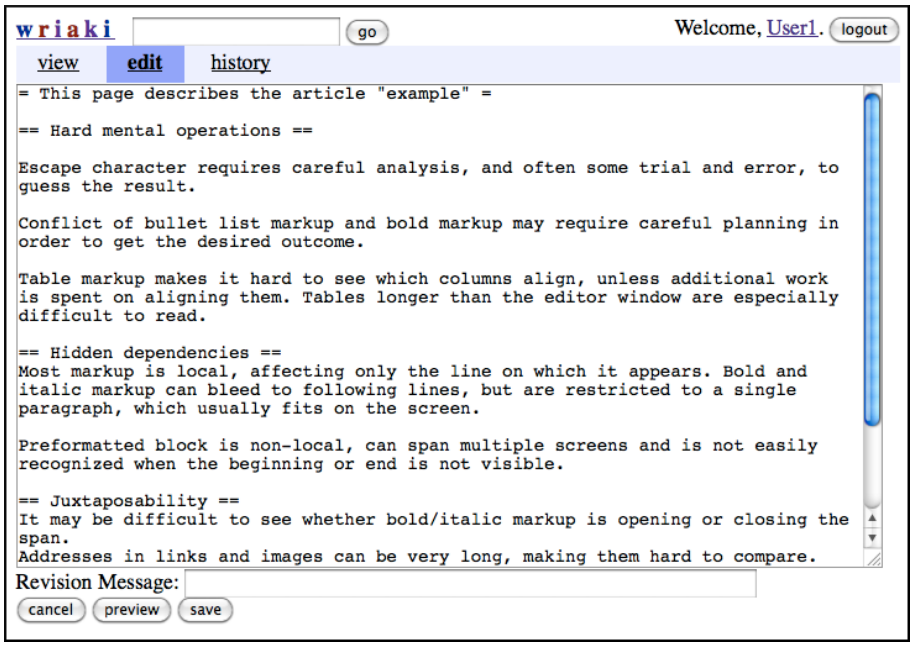

FIGURE 2.3: Edit form of an example article in Wriaki.

2.3.1 Architecture

The architecture of Wriaki is illustrated in fig. 2.4. It is built on top of Webmachine, using it to handle the HTTP application's behavior in a correct way and to structure the application in relation to it. Wriaki is mainly developed using *Erlang*, with small parts in *Javascript* and *Python*. Wriaki allows users to write the wiki articles using the *Creole 1.0* [14] markup language. A parser tool for the *Creole* specification is not available for *Erlang*, and to allow editing on *Creole* format, the *Python Creole 1.0 parser* [15] is used by the Wriaki application. To generate the response HTML pages, Wriaki makes use of the *ErlyDTL* [16] module, which implements the *Django Template Language* [17]. This makes use of DTL templates that can generate HTML documents during execution time allowing the application to generate the different answers. To handle the communication with Riak, Wriaki can be configured to make use of the Protocol Buffers based interface or either the HTTP based one. This communication is always performed through only one Riak node, whose IP address and port should be specified in the configuration file.

2.3.2 Web Resources

As a *RESTful* web application, Wriaki exposes the following resources, which are described in its documentation [18]:

/user – This resource accepts only the *GET* method and provides the form to log in.

/user/<username> – With the method *GET* and without query parameters provides a page with information about the user. With the query parameter **?edit** provides a form to edit the

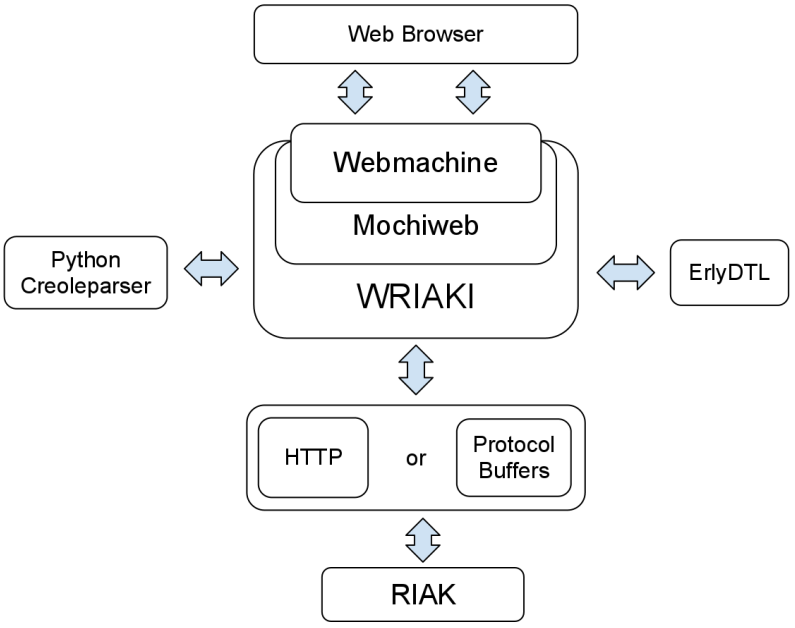

FIGURE 2.4: Wriaki architecture.

user information. By using the *PUT* method it is possible to modify the user data, and by using *POST* users can log in.

/user/<username>/<sessionid> – Provides information about the session for the user logged in. With the method *GET* provides information about the session. Using *DELETE* it removes the session (user logout).

/wiki/<page name> – With the method *GET* without query parameters provides the article page. With query parameter **?edit**, returns an edit form, with **?history**, returns a list of the existing versions of the object, with **?v=<version>**, returns the page version requested, and with **?diff&l=<left_version>&r=<right_version>** returns the difference between the given versions. With the method *PUT*, this resource allows to store a new version of the article or wiki page, and with the method *POST* provides a preview in HTML of the edition, without storing it.

/static/* – This resource only accepts the *GET* method, and provides a way to retrieve specific files. It is used to obtain the *Javascript* and *CSS* code.

2.3.3 Data structure

Wriaki stores data in Riak, and it is organized around five different buckets: *Article*, *Archive*, *History*, *User* and *Session*. The *User* bucket stores the information about the users. The key for the objects on this bucket, which is of course unique for each object, is the *username*. The body of each object is a *JSON*, which stores user information like password or email. The other three buckets are used to store all the information about the articles stored on Wriaki. The

Article bucket stores one object for each page of the Wiki, always the newest one. The key used is just the page name itself, and the body is a JSON which stores the text of the article, the commit message, the time of the edition with precision of seconds and the version hash, a unique number that identifies each version. This hash number, represented in hexadecimal, is generated using the text of the article, the commit message and the time. An interesting point of this bucket is the fact that it is configured to allow simultaneous writes, which allows two or more different users to edit an article concurrently. When this happens, the object stored in the bucket will contain *siblings* instead of just one object, storing all the versions created in parallel, and when retrieving the actual page the user will be warned about all the existing parallel versions. Each article also stores a link to an object of the user bucket, the user who edited it.

The *Archive* bucket is used to store a copy of all existing versions of every page, including the present one, also stored on the *Article* bucket. The key used is the hash number of the version, followed by the name of the article, separated by a dot. The objects store the same body as the ones on the *Article* bucket. The simultaneous writes are not a problem for this bucket, since every version has one object in this bucket, so having two versions created at the same time means just having two new objects in this bucket. The simultaneous editing of the same object never happens, just the simultaneous creation of objects.

Regarding to the *History* bucket, it stores one object for each article, and the key is the same as the one for the *Article* bucket, the page name. These objects store a list of links, each of them pointing to one version of the article, stored in the *Archive* bucket. This bucket is also configured to allow multiple writes. When a user edits an article he also edits the article object in the History bucket to add the new link. If other user edits at the same time the article, generating siblings in the *Article* bucket, will also edit the *History* bucket object to add the link. This will generate siblings in the *History* bucket, since they are editing the same object at the same time, the list of links for the article. Handling this situation is simple, since for every edit operation performed simultaneously, one link is added to the list. Then, each sibling version will have the old links, plus a new one not present in the other simultaneous versions. Performing a union set of this lists will generate the final correct one and solve the conflict. Finally, the *Session* bucket is used to store information about the user sessions, and its data layout is not relevant for the testing model.

An example of this data structure is represented in fig. 2.5. In this representation, two users are registered in the application (*User1* and *User2*), and only two pages have been created. From this pages, '`Welcome`' has been edited two times and '`Pagename1`' three times. To make the representation more clear, the contents of '`Pagename1`' have been omitted in the buckets *Article* and *Archive*. As we can see, the header field of the objects is used to store links to other objects, and the body, which is always a JSON object, to store the data.

We can see as an example a sequence of three edit operations, represented in fig. 2.6, where siblings appear. Consider that *User1* retrieves the edit form for the '`Welcome`' article (/wiki/Welcome?edit). When *User1* is writing his modification, *User2* also retrieves the same form,

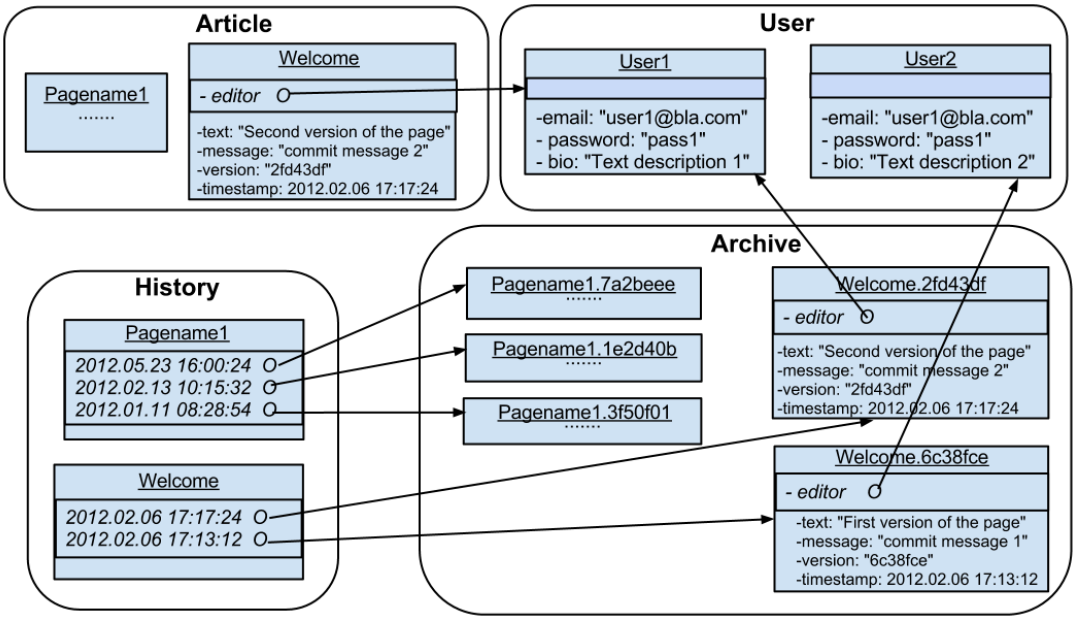

FIGURE 2.5: Wriaki data structure example.

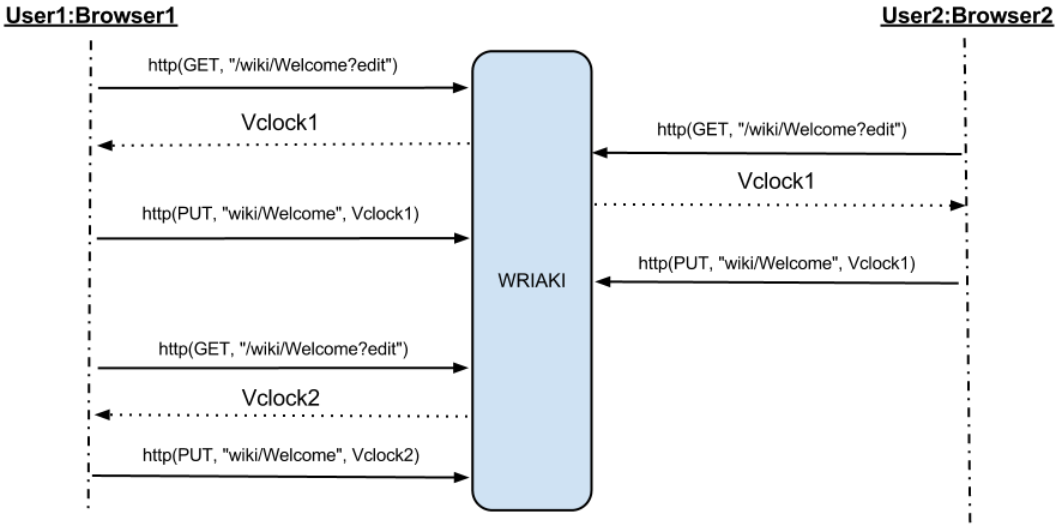

FIGURE 2.6: Possible Secuence of edit operations in Wriaki.

obtaining the same vector clock as *User1*. Then *User1* finishes writing and sends the update, and *User2* does the same after some seconds. The resulting data structure is represented in fig. 2.7.

We can see that siblings have been created in buckets *Article* and *History*. In the *Article* versions, the two new editions are stored instead only one as happens in a normal situation, and in the *History* bucket we have also two objects instead of one, both of them containing all the links stored in the previous object, and each of them also stores a new link to one of the two new objects in the *Archive* bucket.

As it was explained before, Wriaki will automatically handle the conflict in the *History* bucket

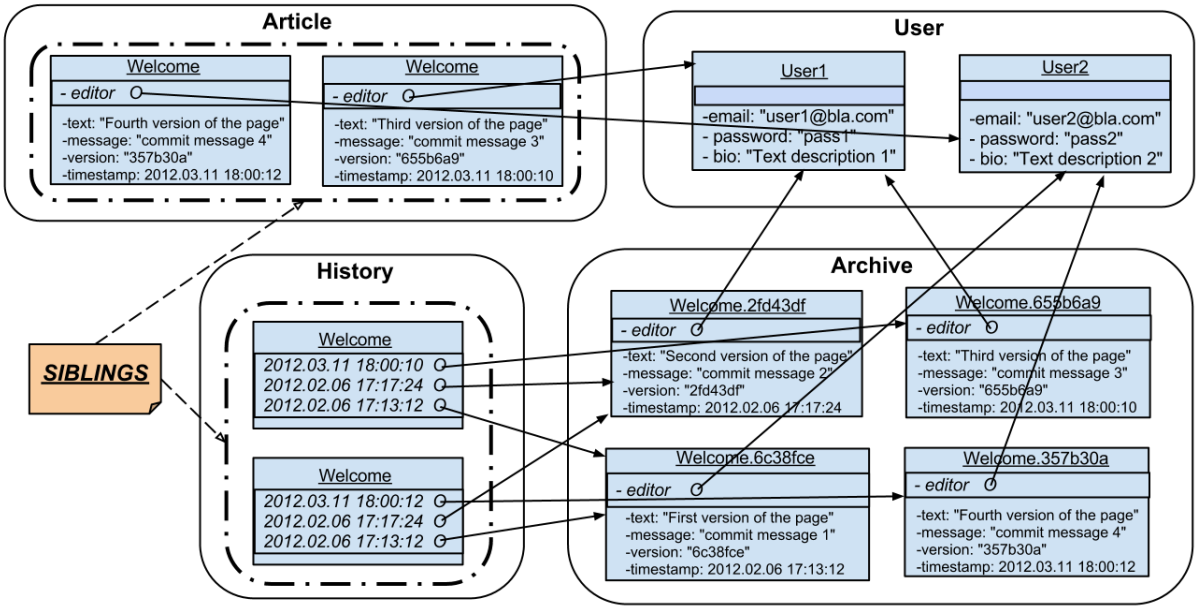

FIGURE 2.7: Wriaki data structure where siblings appear.

by performing a set-union of the lists of links from both objects. The conflict in the *Article* bucket will be solved once a new edit operation of the article is performed using the current vector clock. Having a situation with siblings like in fig. 2.7, when a user retrieves the article page, the most recent version from the ones contained in the siblings will be shown, and also a message warning about the other versions contained in the siblings and providing links to them will be displayed on top of the page. Then, the user can see and analyze all the conflicting versions, and then perform a write like usually though the edit form, which will retrieve the current vector clock.

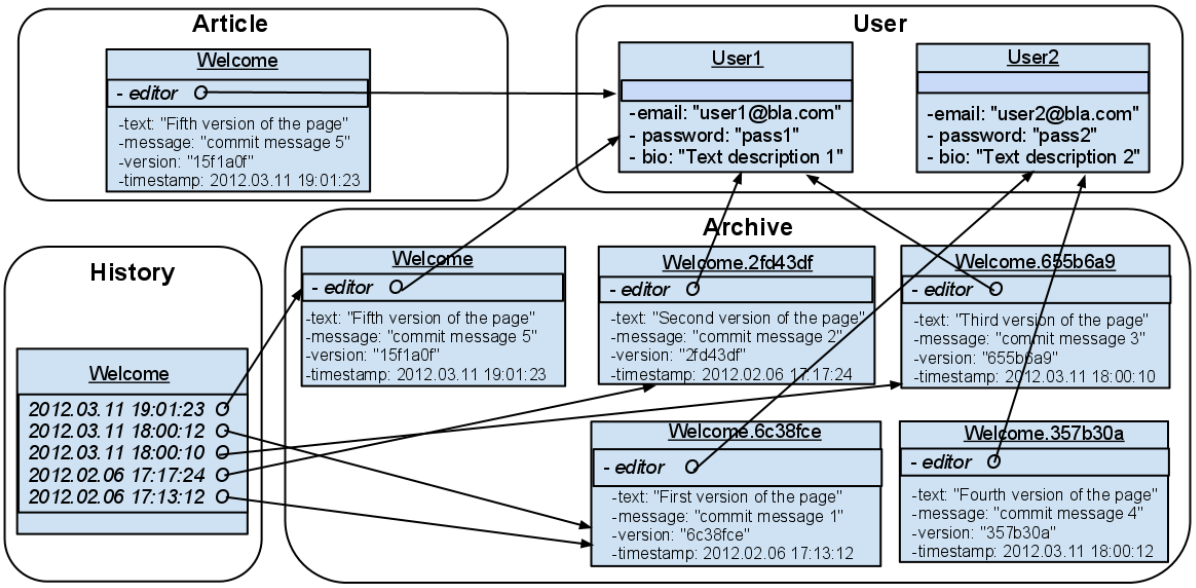

FIGURE 2.8: Wriaki data structure after the fifth write.

In fig. 2.8 we can see the resulting data structure once *User1* has performed the fifth write of the article, using the vector clock obtained when we had siblings in the bucket *Article*. We now have five objects in the *Archive* bucket, one for each edition, and only one in the *Article* bucket. The object in the *History* bucket is the result of the set union of the siblings, containing four links, and with the new link pointing to the fifth edition added. In the *Article* bucket, the siblings have disappeared and now we just have the new edition performed by *User1*.

2.4 Quickcheck

Quickcheck [3] [4] is an automatic testing tool developed in Erlang. It uses an approach called *property-based* testing. Instead of specific tests cases, the user writes a specification for Quickcheck stating the properties that the system under test should always satisfy. Then Quickcheck, based on that specification provides random test case generation and execution, allowing the user to execute a big number of tests in little time. When a failing test case is found, Quickcheck automatically *shrinks* to the smallest possible counterexample, making it easier to understand the reason of the error.

Quickcheck also provides *state machine* based testing, that allows verification of *stateful* systems. By defining function callbacks for the module `eqc_statem` the user can specify:

- The state machine that represents the real *state* of the application under test.
- The operations or *commands* that are executed towards the application and modify its state.
- For each of the commands, its *state transitions* that change the state of the defined state machine.
- Also, for each of this commands, the user can define *preconditions*, which define in which states the commands can be executed or call.
- Finally and also for every command, the *postconditions* that define whether the result is the expected one or not.

By creating a callback module that implements the `eqc_statem` *behaviour*, we can specify an abstract state machine that models the changes on the web application caused by the operations in the test sequences. On the created module it is possible to specify the possible operations or *commands*, the *state transitions*, and the *postconditions* that need to hold after every command execution. A *test case* is a sequence of commands, and given such a module, QuickCheck will be able to generate random sequences of commands or tests, run a certain number of such tests checking that the postconditions always hold and, when finding a failing test case, to automatically *shrink* the sequence of commands to find a minimum failing test case.

Chapter 3

Methodology

The testing of web applications is complex and involves a big number of different methodologies and goals. It can address any non-functional aspects of the application like security testing, load testing or usability testing; or it can focus on the functional aspects. Within the functional testing we can distinguish between *unit* testing, which addresses single elements of the application, and system or integration testing, which considers the complete functionality of the application and is focused in testing complete use cases where several components of it are involved. Looking through the extense literature on the topic, we have found several methods addressing unit testing, but few focusing in more complex functional testing.

We intend to find a testing procedure that can allow us to test the correct behavior of Wriaki, and therefore of any web application with similar characteristics, towards concurrent requests. It is important to notice that the aim is not to provides *unit* testing, since our interest is to test if the application provide correct responses towards complex use cases and, more specifically, towards several of such use cases being performed at the same time. For example, we want to test situations like several users performing edits or requests of the same article at the same time and that in such situations the resulting data structure in Riak is the expected one. Thus, we can say that we are focusing on the system or integration testing of the web application.

3.1 HTTP monitoring tool

To provide such functional testing of the web application, we will use the testing tool introduced in section 2.4. We will model the concurrent web browsers using QuickCheck's state machines to test at the level of Wriaki's web API. For this purpose, the QuickCheck specification needs to be able to generate HTTP requests that modify the *state* of the web application, in the same way as the users do through a web browser. When interacting with the web application using a regular web browser, the Javascript code is executed to generate part of the HTTP requests, thus if we want to completely model the execution of the application it is necessary to execute Javascript code using Erlang and QuickCheck. Considering also that the execution of the Javascript code of

the application is always performed by the web browser, and in consequence different executions may occur on different web browsers, we will use another approach: to directly include in the model the possible HTTP requests generated when executing the Javascript code. Thus, the Javascript code correctness with respect to the browser running on is out of the scope of the testing, focusing on the web application behaviour towards correctly generated HTTP requests. The QuickCheck specification will model the HTTP interface between the web browser and the Wriaki web application, as shown in fig. 3.1.

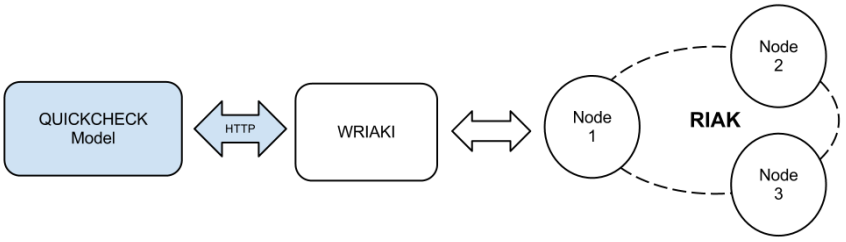

FIGURE 3.1: First test approach.

To do this the HTTP request and responses generated when interacting with the web application will be monitored and analyzed using an external tool to be able to understand their format and meaning, and enable to directly include them in the modeled interface. This approach has some clear issues, like the already mentioned fact that the execution of the Javascript code could be different depending on the web browser under use, or the impossibility of knowing if the obtained set of HTTP requests and responses is complete or not. To deal with such problems, it is possible to run the application using many different web browsers and monitor the traffic to analyze possible differences, and try to ensure that all the use cases are executed by having a good specification of the application.

To analyze the traffic generated by the interaction between the web browser and Wriaki, we will use a tool that can act as a proxy between them and show the details of the HTTP protocol in a clear way. The tool is called Charles [19] and acts as a *proxy*, making it possible to configure the web browser to send the requests to it, which are then sent to the web application. The program provides an interface where it is possible to see all the requests ordered by time and analyze their parameters and content. This makes possible to have a good understanding of their format and be able to implement them in the model. Considering the lack of a good specification for the applications HTTP requests and responses format, the use of the monitoring tool was crucial to find out how to generate them when building the model. When monitoring the HTTP traffic, all possible use cases should be executed. This wont probably be an issue since the Wriaki application is simple and a good specification of the web resources exposed by it is provided, but when following the same procedure on bigger and complex web applications, ensuring that all possible HTTP requests are monitored and thus generated on the model can be a difficult issue.

As a simple example, if we log in a user with username `user1` and password `pass1` in the web application we can see the following HTTP request in the proxy:

```
POST /user/User1 HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: */*
Accept-Language: es-es,es;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: http://127.0.0.1:8000/user
Content-Length: 14
Cookie: session=; username=
Pragma: no-cache
Cache-Control: no-cache

password=pass1
```

From this information we can create functions that perform the requests and that are going to be called by our testing framework. In the login example shown, we can simply define a function that takes the username and password and sends the request:

```
login_user(User, Password, {Browser, Port}) ->
     httpc:request(post, {"http://" ++ ?IP_WRIAKI ++ ":" ++ Port ++ "/user/" ++ User,
     [], [], "password=" ++ Password}, [], [], Browser).
```

Through the proxy we can also see the response provided by the web application. In this example we can see that the expected response from a successful login is `204 No Content`, so we can use this information to create the *postconditions* that are going to check the correctness of the results. In more complex requests might be necessary to parse the HTML code obtained to extract relevant information for the testing procedure, and also to handle some issues related with the cookies.

3.2 Use of Quickcheck

When defining the model it is important to consider the fact that once Riak is up and running, Wriaki stores all the data on it. This means that every time we execute a test, it will make changes in the Wiki, and therefore will store new data in Riak. If we want to have an empty Riak, we just need to stop and restart the three Riak nodes. Even if this seems a simple operation, it is too expensive to do it for every test. Therefore, instead of considering that Riak is empty for every test execution, we consider it to be in a certain state with some data already loaded. The tests are designed to make changes on the web application and check that the application behaves correctly in relation to them. For this to work, the test sequences need to be independent of the initial state of Wriaki and the model needs to be adapted to this situation.

As we already said, we are going to use the `eqc_statem` module to define our model and for this we need to define the required *callback* functions. After having analyzed the format of the HTTP messages in Wriaki, we will have a list of possible requests that the application can handle that are going to be our *commands*. We use the *state* definition to store relevant data related to the state of our application and the *state transitions* to define the relevant changes that the requests performed do in our state. Then, we can use the information stored in the defined state to use the *preconditions* to know when the requests or commands can be included and through the *postconditions* check if the response is the expected one. To do this last check we will need to parse the HTML code of the body of the response message obtained to be able to extract the relevant information that we need to check towards the one stored in the state to know whether the response is correct or not.

Once the model is defined, we can initialize as many web browsers as we want and use them to send the requests. Quickcheck also provides a simple way to execute test cases in parallel, which will allow us to have several of this web browsers executing test cases concurrently.

Chapter 4

Implementation

The Wriaki version under use is available at Github [18], and the last update at this moment was performed on 6th of October, 2011. The version of Riak installed is 1.0.3 and Wriaki is configured to connect to it using protocol buffers. Riak is used with a default configuration of three nodes using *Memory* as the *back-end* storage system, which stores all the data on memory instead of disk.

4.1 Model definition

As it was explained in section 2.4, to create the specification for Quickcheck we need to define a number of function callbacks for the module `eqc_statem`. The first thing that we need to define in our module is a data structure that represents the state. This can be done using records:

```
-record(state, {pages, browsers, users}).
-record(user, {name, pass}).
-record(version, {number, editor, text, rev_msg, vclock_used}).
```

The state is a record with three fields. The first of them, `pages` is a list of tuples containing a String with the page name and a list of records of type `version`, used to represent a version of a page. Each of these `version` records consists of a `number`, used to represent the order of the different versions with increasing values, the `editor` or user who created the version, the content of the version (`text`) and the revision message for the edition (`rev_msg`). Finally, `vclock_used` stores a copy of the vector clock used for that edit operation, which is just an invariant needed to perform the edit operations. From the remaining two fields, `browsers` stores the initialized browsers that can be used to send requests to the applications, and `users` stores the information about the different users that can log in into Wriaki.

Following this data structure, it is necessary to define the function `initial_state/0`, that will give initial values to the state. The defined function just introduces values for the pages that

already exist in Wriaki and users that are already signed up. This initial state implies that for the testing to work, three users ('`User1`', '`User2`' and '`User3`') need to be already signed up in Wriaki with their correct passwords ('`pass1`', '`pass2`', '`pass3`'). Also the web pages present on the initial state function need to have at least a first version existing in Wriaki. During the test execution, the changes on the state will be reflected in this presented data structure, and they will be defined in the `next_state/3` function. The next function that we need to define is the *command generator*, `command/1`, which generates a proper command, considering the actual state, to be added to the test sequence:

```
command(S) ->
 oneof(
   [{call,?MODULE,login_user,[anyuser(S), elements(available_browsers(S))]}
    || available_browsers(S) /= [] ] ++
   [{call,?MODULE,logout_user,[elements(S#state.browsers)]}
    || S#state.browsers /= [] ] ++
   [{call,?MODULE,request_page,[page(), elements(browsers())]}]).
```

Each one of the commands present in the `command/1` function calls a function that performs the HTTP request. For example, the `login_user` command calls a function that performs a POST request to login the specified user. For now on, the function `command/1` generates three kinds of commands, that reproduce the actions of a user log in, log out and requesting a page. It is important to notice that we need to define the number of browsers available in the function `browsers/0`. The number of browsers relate to the number of possible users logged in at the same time. The function `available_browsers/1` checks the state and returns a list of the browsers in which a user is not logged in at that moment, and therefore are available for users to log in. This commands are only included in the test sequence if their *precondition*, which depends on the state, holds. This preconditions are defined in the function `precondition/2`, which typically has one clause for each possible command, and this is included only if its clause holds. For this three defined commands, the precondition function looks like this:

```
precondition(S,{call,?MODULE,request_page,[Page, _Browser]}) ->
     lists:member(Page, edited_pages(S));
precondition(S,{call,?MODULE,login_user,[{_Name, _Pass}, Browser]}) ->
     lists:member(Browser, available_browsers(S));
precondition(S,{call,?MODULE,logout_user,[Browser]}) ->
          lists:member(Browser, S#state.browsers);
precondition(_S,_) -> true.
```

Taking a look to the function, we can see for example that we only request pages that have been already edited, which means that `lists:member(Page, edited_pages(S))` holds true, and that we only log in a user if the browser is available. As it was already mentioned, we also need to define the `next_state/3` function, which modifies the state after every command is executed. For the three first commands defined, we have the following function:

```
next_state(S,_V,{call,?MODULE,login_user,[{Name, _Pass}, Browser]}) ->
       S#state{browsers = [{Browser, Name, no_page_loaded} | S#state.browsers]};
next_state(S,_V,{call,?MODULE,logout_user,[Browser]}) ->
   S#state{browsers = lists:delete(Browser, S#state.browsers)};
next_state(S,_V,{call,_,_,_}) -> S.
```

Obviously, the `request_page/2` command doesnt change the state, and the other two commands just add or delete the user-browser tuple on the state. It is also necessary to define the *post-condition* for every command, which are going to check that the output is actually the correct one and therefore allow the testing model to detect the errors. They are defined in the function `postcondition/3`, being possible to write one clause for every command. As an example, for the `login_user/2` command, we just check that the response code is the expected one when the request succeeds:

```
postcondition(_S,{call,?MODULE,login_user, [{_Name, _Pass}, _Browser]},V) ->
     get_response_code(V) == 204;
```

Finally, to be able to run the tests, we need to define a *property* that generates the commands, runs them, and checks that the postconditions hold at all times:

```
        prop_wriaki() ->
                ?FORALL(Cmds,commands(?MODULE),
                     begin
                        [start_browser(A) || A <- browsers()],
                        {H,S,Res} = run_commands(?MODULE,Cmds),
                        [logout_user(A) || A <- S#state.browsers],
                        [stop_browser(B) || B <- browser()],
                        ?WHENFAIL(io:format("~p\n~p\n~p\n",[H,S,Res]), Res==ok)
                      end).
```

Now it is possible to run the test by executing `eqc:quickcheck(wriaki_test:prop_wriaki())` in the Erlang shell. Several sequences of 100 tests pass without finding any error.

4.2 Support for editing of articles.

The next step will be to add support for editing of articles in the model. For that, we add two new commands to the `commands/1` function:

```
[{call,?MODULE,load_edit_page,[page(), elements(S#state.browsers)]}
      || S#state.browsers /= [] ] ++
[{call,?MODULE,edit_page,[elements(browsers_loaded_page(S)), new_text(), revision_msg()]}
      || browsers_loaded_page(S) /= [] ]
```

The `load_edit_page/2` command does the same as the user requesting the edit form of a page, it retrieves the current vector clock that will be necessary when performing the write. This needs to be done by a browser in which a user is logged in, and the `next_state` clause for this command will store this vector clock in the state, associated with the user and browser that made the request. The `edit_page/3` command does the same as the user pressing save after writing the text in the edit form of an article, it stores the new version in Wriaki. The write is performed using a vector clock, that has previously been retrieved by a call to the `load_edit_page/2` command. Thus, the call needs to be done by a browser that has a user logged in and a vector clock associated, which is provided by the function `browsers_loaded_page/1`. The `next_state` for this command creates a new version record, including details like vector clock used, text, user and commit message, and stores it in the state:

```
next_state(S, _V, {call, ?MODULE, edit_page, [{_Browser, User, {Page, _VC}}, Text, Msg]}) ->
    {Page, Versions} = lists:keyfind(Page, 1, S#state.pages),
    New_versions =
      [#version{number=length(Versions)+1, editor=User, text=Text, rev_msg=Msg}] ++ Versions,
    S#state{pages = lists:keyreplace(Page, 1, S#state.pages, {Page, New_versions})};
```

For now on, the postcondition for the edit command only checks that the response code is the expected one, 204, and the text for each edit operation is selected from a simple function, which randomly selects three possible short strings.

4.3 Checking data correctness

After adding this new commands, the generated test sequences pass without any problem detected, which means that edit operations are performed and the correct response code is generated. But we are still not checking if the updates are performed in a correct way, and that the new data follows the structure explained in section 2.3.3 at page 8. In a concurrent update, if *User1* edits a page, and then *User2* does the same, when *User1* has a look to the page, he will be confronted with updates that he hasn't performed himself. This represents a challenge for testing, and it is the reason to store a version record for each edition in the state, enabling to know the edit requests performed by the different browsers and users, and their respective order. This allows us to model and test the correctness of concurrent updates. Having this information already stored in the state, we need to retrieve the relative information from Wriaki to check its correctness. For that we can use two resources. The first one is the page version (**/wiki/<pagename>?v=<version>**), which allows us to retrieve any old version we want, but for doing that we need to know the hash number that identifies the version, and we dont have it, since when requesting an edit Wriaki doesnt provide it in the response, which makes the testing procedure more complicated. This fact makes necessary to use the other resource, the page history list (**/wiki/<pagename>?history**), which displays information about all the commits, as we can see in fig. 4.1. To retrieve this history list we add a new command to the `commands/1` function:

```
[{call,?MODULE,check_page_history,[elements(edited_pages(S)), elements(browsers())]}
    || edited_pages(S) /= [] ]
```

FIGURE 4.1: Page history list view for the article Welcome.

The function `check_page_history/2` receives an already edited page and a browser, and retrieves the history list for that page using the specified browser. This function doesnt change the state and therefore its `next_state` simply returns the state unmodified. Using the postcondition, we need to check both, that the displayed information follows correctly the data structure described in section 2.3.3, and also that is equivalent to the information stored in the state of the model. The first thing that is interesting to check is whether the hash numbers are always unique:

```
postcondition(S,{call,?MODULE,check_page_history,[Page, _Browser]}, V) ->
    {Page, Versions} = lists:keyfind(Page, 1, S#state.pages),
     ListHash = [ A || {A,_, _} <- lists:sublist(V, erlang:length(Versions))],
     ListHash -- lists:usort(ListHash) == [];
```

Surprisingly, when executing the tests the postcondition evaluates to false after 7 tests, but we seem not to shrink to the smallest possible *counterexample*. By repeating the test in the shell, we observe different shrinking behaviours, normally a sign that the test case does not fail deterministically. We suspect a concurrency error to limit the shrinking possibilities and therefore we introduce the `?ALWAYS` macro. By using this macro, we run each test a fixed number of times during the shrinking (10 in our case) to check whether it fails at least once in those cases. Doing so helps to find a minimum counterexample:

```
[{set,{var,14}, {call,wriaki_eqc, login_user,[{"User1","pass1"},browser2]}},
 {set,{var,17}, {call,wriaki_test_eqc,load_edit_page,
         ["Welcome",{browser2,"User1",no_page_loaded}]}},
 {set,{var,18}, {call,wriaki_test_eqc,edit_page,
         [{browser2,"User1",{"Welcome",{var,17}}},"text content 1","commit_messageA"]}},
 {set,{var,19}, {call,wriaki_test_eqc,edit_page,
         [{browser2,"User1",{"Welcome",{var,17}}}, "text content 1","commit_messageA"]}},
 {set,{var,22}, {call,wriaki_test_eqc,check_page_history,["Welcome",browser1]}}]
```

4.4 Analyzing and fixing the error found

From the counterexample given we can observe that both editions have been done by the same user, using the same vector clock and with the same text and same commit message. Taking a look at Wriaki's source code we can find the function where the hash numbers are generated in the file `article.erl`. The function is `update_version/1`:

```
update_version(Article) ->
   {MS, S, _US} = now(),
   TS = 1000000*MS+S,
   wobj:set_json_field(wobj:set_json_field(Article, ?F_TS, TS), ?F_VERSION,
         list_to_binary(mochihex:to_hex(erlang:phash2({get_text(Article),
           get_message(Article), TS})))).
```

As we can see, the hash numbers are generated using *three* elements: the text of the edition, the commit message and a timestamp (TS). Obviously, when the three elements are the same it will generate the same hash number. In our test case, which is a valid sequence, the text and the commit message are the same. Since we have duplicated hash numbers, the problem is that the timestamps are also the same, which is a clear error. The timestamp is generated using the function `now/0` which guarantees that subsequent calls always return continuously increasing values, but the *microseconds* are ignored. The consequence is that when *any* two users perform two writes during the same second, with the same commit message and the same text, Wriaki will generate duplicated hash numbers.

The problem can be illustrated using the data structure presented in section 2.3.3. If we consider a *Welcome* page empty and only one user registered in the application, the sequence presented in the counterexample will lead to the situation shown in fig. 4.2. As we can see, *User1* has performed two writes using the same vector clock, which leads to the creation of siblings in the Article bucket. It also creates siblings in the History bucket, but this is solved automatically and the figure shows the resulting unique object in the History bucket. Both writes have been done during the same second, and therefore they generated the same hash number. We can see that both objects contained in the siblings have the same hash number. In the Archive bucket we only have one element instead the two expected. This is because after the first write the first element is created with key '`Welcome.2eb3a29`' . Then, when the second write is performed, Wriaki creates a new object in the Archive bucket, but using the same key as before, so what it does instead is to *overwrite* the previous object. As a consequence of this we can also see that both links from the object in the History bucket point to the same element in the Archive bucket, which is supposed to never happen.

In the counterexample provided, all the actions are performed by the same user. This error can be reproduced by a real user of the application that accidentally, when performing an edit operation, presses the *save* button two times. Knowing the existence of the error, If we do that on the browser we see that it is possible to provoke such error manually, however to find it

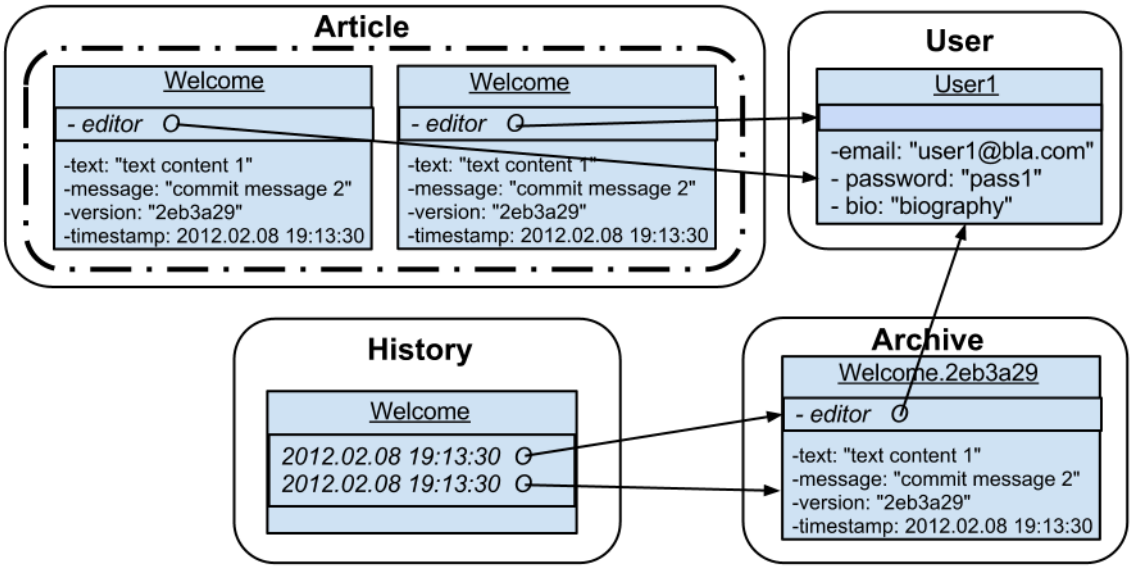

FIGURE 4.2: Data structure showing the hash error.

through regular testing is difficult, considering that only happens under specific conditions that are very unlikely to be reproduced when testing manually. Using QuickCheck makes possible to generate lots of random sequences, simulate high concurrency and perform many requests in little time, which made possible to find the error. It is also important to notice that it was possible to find it since the text and commit messages were generated choosing from small lists of possible strings. This was just a first approach, but its simplicity lead to several commands with the same parameters that provoked the error. Having used complex generators that create random strings, probably would have been impossible to find it. As it was explained before, the error can also be provoked by two different users that simply make an edit operation during the same second with the same parameters, but with the actual model, QuickCheck will provide the error sequence caused by only one user, since the shrinking procedure searches for the *shortest possible counterexample*. To provoke the error with two different users, we can change the precondition clause of the `edit_page/3` command to force consecutive edit operations to be always performed by different users:

```
precondition(S,{call,?MODULE,edit_page,[Browser, _Newtext, _Message]}) ->
   lists:member(Browser, browsers_loaded_page(S)) andalso
    begin
      {_BrowserName, User, {Page, _Vclock}} = Browser,
      {Page, Versions} = lists:keyfind(Page, 1, S#state.pages),
      [] == [A || A =  #version{number=B, editor=C}  <- Versions, B == length(Versions),C == User]
    end;
```

After this change, it shrinks to a counterexample where two different users perform the edit operations. As it was expected, the sequence is longer than the previous one:

```
[{set,{var,1}, {call,wriaki_eqc,login_user,[{"User1","pass1"},browser2]}},
```

```
{set,{var,4}, {call,wriaki_eqc,login_user,[{"User2","pass2"},browser3]}},
{set,{var,7}, {call,wriaki_eqc,load_edit_page, ["Welcome",{browser3,"User2",no_page_loaded}]}},
{set,{var,9},{call,wriaki_eqc,edit_page,
       [{browser3,"User2",{"Welcome",{var,7}}},"text content 1","commit_messageA"]}},
{set,{var,11}, {call,wriaki_eqc,load_edit_page,["Welcome",{browser2,"User1",no_page_loaded}]}},
{set,{var,12},{call,wriaki_eqc,edit_page,
      [{browser2,"User1",{"Welcome",{var,11}}}, "text content 1","commit_messageA"]}},
{set,{var,14}, {call,wriaki_eqc,check_page_history,["Welcome",browser1]}}]
```

Analyzing Wriaki's source code further, we can see that the generated timestamps are used to order the different editions according to the time they were created, and therefore the application should not be able to order correctly editions created during the same second. To check whether this is true or not, we change the model to send revision messages that identify each edition, meaning that the first edition of a sequence will have the string '`edition1`' as its revision message, the second one the string '`edition2`' and so on. This will make impossible to generate repeated hash numbers since the revision messages are never going to be repeated. The postcondition clause is changed to check if the order presented in the history list is the same as the one stored in the state:

```
postcondition(S,{call,?MODULE,check_page_history,[Page, _Profile]}, V) ->
     {Page, Versions} = lists:keyfind(Page, 1, S#state.pages),
     List = lists:sublist(V, erlang:length(Versions)),
     ListHash = [ A || {A,_, _} <- lists:sublist(V, erlang:length(Versions))],
     ListHash -- lists:usort(ListHash) == [] andalso compare_versions(Versions, List);

compare_versions([], []) -> true;
compare_versions([Version | T1], [{_VersionHash, UserName, RevMessage} | T2])
  when  (Version#version.editor == UserName) and (Version#version.revision_message == RevMessage) ->
       compare_versions(T1, T2);
compare_versions(_,_) -> false.
```

As expected, executing the tests the postcondition fails whenever we have editions created in the same second. The problem is easy to see after the test execution by retrieving the history list, as we can see on the left side of fig. 4.3. The revision messages performed during the same second are wrongly ordered. The error in Wriaki can be easily fixed by using a timestamp that considers the microseconds instead, in the already mentioned function `update_version/1`:

```
{MS, S, US} = now(),
TS = 1000000000000*MS + 1000000*S + US,
```

It is also necessary to change the function `format_time/1`, in the module `article_dtl_helper.erl`, to adapt it to the new timestamps. Having fixed the error and recompiled Wriaki, all the tests

executed pass without having either duplicated hash numbers or wrong order in the history list. On the right side of fig. 4.3 we can see an example of the history list after the error has been fixed. From now on, all the tests are going to be performed using instances of Wriaki where the described error is fixed.

l	r	version	time	editor	message
◉	○	5857672	2012.03.13 13:37:34	User1	edition3
○	◉	44c6e22	2012.03.13 13:37:34	User1	edition1
○	○	3f26f71	2012.03.13 13:37:34	User1	edition4
○	○	22351eb	2012.03.13 13:37:34	User1	edition2
○	○	78f734a	2012.03.13 13:37:33	User1	edition2
○	○	7016904	2012.03.13 13:37:33	User1	edition1
○	○	6fdc394	2012.03.13 13:37:33	User1	edition2
○	○	75ad4d2	2012.03.13 13:37:33	User1	edition4
○	○	72c8e8e	2012.03.13 13:37:33	User1	edition1
○	○	7323ede	2012.03.13 13:37:33	User1	edition3
○	○	6df39f1	2012.03.13 13:37:33	User1	edition3
○	○	42e7f75	2012.03.13 13:37:33	User1	edition1
○	○	6e61ead	2012.03.13 13:37:33	User1	edition3
○	○	5bde05d	2012.03.13 13:37:33	User1	edition1

l	r	version	time	editor	message
◉	○	24b31c8	2012.03.12 15:39:15	User3	edition1
○	◉	756a824	2012.03.12 15:39:15	User3	edition1
○	○	32bac1b	2012.03.12 15:39:06	User2	edition1
○	○	4411684	2012.03.12 15:39:04	User2	edition3
○	○	5953ef5	2012.03.12 15:39:04	User2	edition2
○	○	80d902	2012.03.12 15:39:03	User2	edition1
○	○	6b4280c	2012.03.12 15:39:02	User1	edition3
○	○	20cb012	2012.03.12 15:39:02	User1	edition2
○	○	142a796	2012.03.12 15:39:02	User1	edition1
○	○	4083439	2012.03.12 15:39:02	User3	edition1
○	○	577e977	2012.03.12 15:39:00	User1	edition4
○	○	7ba00bd	2012.03.12 15:39:00	User2	edition3
○	○	132c19	2012.03.12 15:39:00	User2	edition2
○	○	326238c	2012.03.12 15:38:59	User2	edition1

FIGURE 4.3: View of the history list page before fixing the timestamps (left) and after (right).

4.5 Testing siblings conflict resolution

We still need to check whether the siblings creation and handling, when we have concurrent writes, works as expected according with the structure explained in section 2.3.3. When we have a situation on an article where siblings appear and we request the page for that article we will get one of the versions contained in the siblings, and a message on top warning about the simultaneous versions also contained in the siblings, as we can see on fig. 4.4.

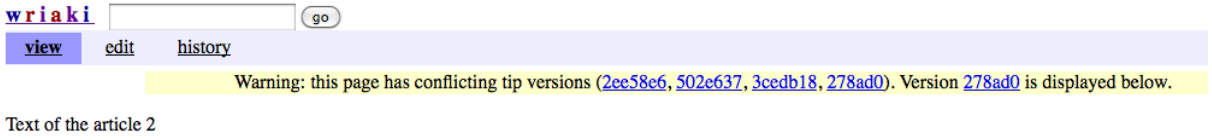

FIGURE 4.4: Page view for an article when having siblings.

The information provided are the hash numbers of all the versions contained in the siblings, between brackets, and the hash number of the shown version. Thus, it is easy to parse the HTML code and obtain the number of siblings that the web application has registered. What we want to do is to check if this is the same number of siblings that we should have according to our model. As we have explained, Riak uses Vector Clocks to identify the concurrent writes and to create the siblings, so if we want to do the same we need to understand and implement the used Vector Clocks algorithm, something that is extremely complex and therefore not possible. We consider the vector clocks as simple *invariants*, and the only thing we can do with them

is to check whether they are the same or not. However, it is possible to know the number of siblings without implementing the algorithm. For this we need to have two elements stored in the state. First, a copy of each vector clock generated during the test execution. Every time we do an edit operation, a new vector clock is generated, and to retrieve it we need to modify the `edit_page/3` command to request it after the edit operation is performed (this makes the command not atomic). The second information that we need is the vector clock used for each edit request, which is already always stored in the state. Having all this information, we can represent it using just a list of numbers, with the same length as the number of editions, where every element represents an edition and they are ordered from the newest to the oldest from left to right. The number itself represents the vector clock used for that edition, so the vector clock before the first edition is tagged with the number 1, the one before the second edition and after the first edition with the number 2 and so on. If we take the sequence of editions presented in fig. 2.1 at page 5 , it will result in the following list:

```
[1,3,3,2,1]
```

As it was already explained in section 2.1, this sequence of edit operations will lead to the creation of 3 siblings. The length of the list is 5, which is the number of edit operations performed. We can see that the third and fourth edit requests were done using the same vector clock, the number 3, and this will lead to have already 3 siblings. Since the last write was performed with the vector clock 1 which is a stale or old one, it will add one more sibling and at the end of the sequence we have 3 siblings. For example, if instead of a 1 for the last edition we have a 5, we would have 0 siblings, since in that case the last edit operation is done with the current vector clock, the one after the fourth write. The function `check_number_siblings/1` receives a list of editions and provides the expected number of siblings:

```
check_number_siblings(ListEditions) ->
      case check_number_siblings_aux(ListEditions) of 0 -> 0; N -> N + 1  end.

check_number_siblings_aux([]) -> 0;
check_number_siblings_aux([_]) -> 0;
check_number_siblings_aux([H|T]) when  H >= length(T) + 1 -> 0;
check_number_siblings_aux([_H|T]) ->  1 + check_number_siblings_aux(T).
```

The new postcondition for the `request_page/2` command, where the `get_conflicting_versions/1` function parses the HTML code and gets the number of siblings according to Wriaki, is as follows:

```
postcondition(S,{call,?MODULE,request_page,[Page, _Browser]},V) ->
    case get_response_code(V) of
       200 -> {Page, Versions} = lists:keyfind(Page, 1, S#state.pages),
```

```
            Vclocks = [ VC || #version{vclock_used = VC}  <- Versions],
            Aux = (fun(Vclock, State_vclocks) ->
                     {Vclock, Number} = lists:keyfind(Vclock, 1, State_vclocks),
                     {Number, State_vclocks} end),
           {EditionsNumbered, _Acc} = lists:mapfoldl(Aux, S#state.vclocks, Vclocks),
           case has_conflicting_versions(V) of
                   false -> check_number_siblings(EditionsNumbered) == 0;
                   true ->  {ConHashes, _HashDisplayed} = get_conflicting_versions(V),
                            check_number_siblings(EditionsNumbered) == length(ConHashes)
           end;
     _ -> false
end;
```

After the changes in the model, several sequences of tests pass without finding any error in Wriaki.

4.6 Testing with two instances of Wriaki

Until now the tests have been executed using the structure presented in fig. 3.1 at page 14. However, considering the distributed nature of Riak, it should be possible to run more than one *instance* of Wriaki at the same time, each of them connecting to a different Riak node, which makes the application very scalable. To test if this works as expected, we change the configuration to the one shown in fig. 4.5, having two instances of Wriaki, connecting to two different nodes of Riak.

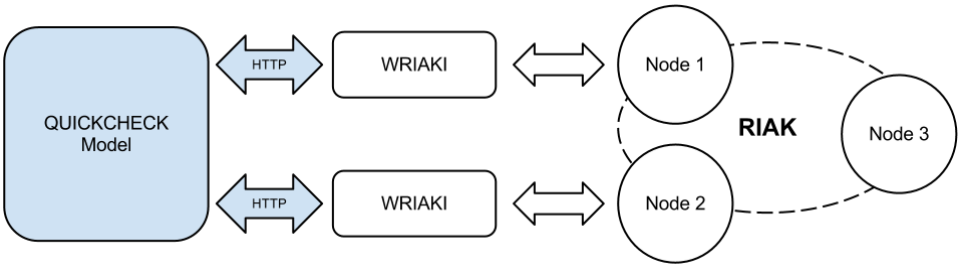

FIGURE 4.5: Test approach with two instances of Wriaki running.

In our test model we focus on the correctness of the data structure and its content regarding to the executed test cases. When we have only one instance of Wriaki running, all the operations that affect the data structure like key generation, siblings conflict resolution or value updates are performed by it. However, if we add one or more instances running in parallel, all this actions are also performed by the other instances at the same time over the same data, and therefore errors that will never happen with only one instance running can appear with several due to errors in the design. As an example of one of such errors we can consider the problem described in section 4.4. Once we have fixed the error, the hash numbers are generated using the function `now/0`, which is supposed to return always increasing values, and therefore we always

have different hash numbers. However, if we run a configuration like the one in fig. 4.5, we can have the two instances running in different machines which can eventually return the same hash number and introduce an inconsistency in the data. Even if this could happen, the probability is very low and we don't expect to detect this problem with our test approach.

To support this new structure, we need to make minor changes in the model. Until now, our model was designed to send all the requests to the same instance of Wriaki, that was listening in an specific IP and port. Considering that now we have two instances of Wriaki listening in different ports, it is necessary to specify for each browser included in the model to which port it should send the requests, and also to modify every command function to receive the port number and to send the requests to it. Once these changes are done, we can easily change the number of browsers in the model and the ports associated to each of them. To test the configuration shown in fig. 4.5 we configure it with six browsers, the first three browsers send the requests to the first instance of Wriaki and the remaining three to the second instance. Running the tests with this new setup, no errors are detected.

4.7 Parallel testing

Even if during our testing we have tested concurrent writes retrieving vector clocks and performing edit operations separately, all the requests in every test sequence were executed sequentially. In a real environment, several users will be sending requests to the application from different browsers running at the same time, and such requests will be received by the web application in an interleaved way. Having the possibility of reproducing such a situation with our testing model could allow us to detect errors caused by race conditions that would be impossible to find by simply executing sequences of sequential tests like we have done until now. QuickCheck supports parallel testing and to generate and run parallel test sequences using our model only requires to write a new property. A *parallel test sequence* consists of a list of commands that are going to be run sequentially at the beginning, followed by one or more lists of commands that are going to be run in parallel. However, there is a problem related with our model, where every command is an HTTP request performed by one of the browsers. Each of these browsers that our model handles performs its requests sequentially, so if we have a situation where requests performed by the same browser appear in different parallel branches, they will be sequentialized by the browser itself, so the achieved parallelism may not be optimal. If we generate some samples of parallel test cases using `eqc_gen:sample/1` we can see that the problematic pattern appears sometimes:

```
{[{set,{var,1}, {call,wriaki_eqc,login_user,[{"User1","pass1"},{browser2,"8000"}]}},
  {set,{var,2}, {call,wriaki_eqc,load_edit_page,
            ["Welcome",{{browser2,"8000"},"User1",no_page_loaded}]}},
  {set,{var,3}, {call,wriaki_eqc,edit_page,
            [{{browser2,"8000"},"User1",{"Welcome",{var,2}},0},"Text of the article."]}},
  {set,{var,4}, {call,wriaki_eqc,logout_user, [{{browser2,"8000"},"User1",{"Welcome",{var,2}}}]}}],
```

```
 [[{set,{var,5}, {call,wriaki_eqc,login_user,[{"User2","pass2"},{browser1,"8000"}]}},
   {set,{var,6},{call,wriaki_eqc,logout_user, [{{browser1,"8000"},"User2",no_page_loaded}]}},
   {set,{var,7}, {call,wriaki_eqc,login_user,[{"User2","pass2"},{browser2,"8000"}]}},
   {set,{var,10}, {call,wriaki_eqc,login_user,[{"User1","pass1"},{browser1,"8000"}]}}],
  [{set,{var,8}, {call,wriaki_eqc,check_page_history,["Welcome",{browser2,"8000"}]}},
   {set,{var,9}, {call,wriaki_eqc,request_page,["Welcome",{browser1,"8000"}]}},
   {set,{var,11}, {call,wriaki_eqc,check_page_history,["Welcome",{browser1,"8000"}]}}]]}
```

As we can see, this test case consists of an initial list of commands that are going to be executed sequentially, the first four of them, and a list of lists of commands to be executed in parallel. In the shown example we have two lists of commands. As we can see, in the first of them there are requests to be performed by two browsers (`browser1` and `browser2`). In the second one, we see requests also performed by the same browsers (`browser1` and `browser2`). We want to only execute sequences in parallel where the browsers that are used in each of them don't appear in the other ones. This is because every browser executes its requests sequentially, and thus having the same browser appearing in different parallel test cases makes the achieved parallelism not good, since the execution of one parallel brach is conditioned by the other one. When having different web browsers in all the parallel branches we now for sure that we are executing the requests in parallel.

The random generation of tests sequences from our model is handled by QuickCheck, and there is no easy way to change or influence the procedure. However, a possible way to solve the issue is by using the macro `?IMPLIES`, that allows us to discard the tests cases that we don't want to execute. By doing this we still generate problematic test cases, but we only execute the ones where the explained pattern doesn't appear. To be able to discard the test cases where the pattern appears, we define the function `check/1`, which receives the list of sequences to be executed in parallel and returns false if the pattern is found:

```
check([]) -> true;
check([_]) ->true;
check([H|T]) -> Browsers = lists:filter(appears(H), browsers()),
                           case lists:dropwhile(aux(Browsers), T) == [] of
                                 false -> false;
                                 true -> check(T)
                           end.

aux(Browsers) -> fun(ListCommands) -> lists:filter(appears(ListCommands), Profiles) == [] end.

appears(ListCommands) -> fun(Browser) ->
                     string:str(lists:flatten(io_lib:format("~p",[ListCommands])),
                     lists:flatten(io_lib:format("~p",[Browser]))) /= 0 end.
```

The final property for parallel testing, which discards the problematic test cases, is as following:

```
prop_wriaki_parallel() ->
```

```
?FORALL(Cmds,parallel_commands(?MODULE),
     ?IMPLIES(begin
                    {_Sequential, Parallel} = Cmds,
                    check(Parallel)
                    end,
                    begin
                        [start_browser(A) || {A,_} <- browsers()],
                        {H,NotS,Res} = run_parallel_commands(?MODULE,Cmds),
                        [stop_browser(B) || {B,_} <- browsers()],
                        ?WHENFAIL(io:format("~p\n~p\n~p\n",[H,NotS,Res]), Res==ok)
                    end)).
```

The percentage of discarded test cases depends of the number of browsers used. The more browsers we have, the smaller the probability of having the same browser appearing in two different branches. To check this we generate a sequence of 1000 tests and using the function `eqc:collect/2` we can get the percentage of discarded test cases, which are the ones that don't satisfy the `check/1` function presented above. Having just two browsers, 40.2% of the generated test cases are discarded. If we use three browsers, this percentage decreases to 37.1%, with six browsers to 27.2% and having ten browsers only 17.8% of the test cases are discarded. Having defined the property presented above, several sequences of 1000 tests, in which the problematic test cases have been discarded, are run without detecting any error in the web application.

4.8 Further testing.

In the previous sections we have described only the most relevant parts of the model of the web application. The model has been developed further that what is shown, adding several commands to provide testing of other functionalities of the application and the remaining resources described in section 2.3.2. With this further additions many sequences of tests were executed using different configurations, but it was not possible to find new errors in the web application. Besides the error explained in section 4.4, this shows the robustness of the application under test.

Chapter 5

Related Work

A methodology for testing traditional database systems that also makes use of Quickcheck has been developed [20]. The methodology is based on defining constraints present on the database under test, and then building a model of it based on state machines and generate random queries that modify the data, checking that the constraints hold after every test. This procedure has been applied successfully in a real database system belonging to a financial company [21]. At first, we considered the possibility of using this approach to test the correctness of the data structure of our application. However, the fact that in Riak the data consistency is not guaranteed and thus it does not provide *transactions* makes complicated to apply the methodology to Riak based systems.

Regarding to web testing, an analysis of its peculiarities, different testing approaches and interesting research done in the topic can be found in [2]. The paper analyzes the peculiarities of web applications and their heterogeneous nature, which makes them difficult to test. It describes different types of testing that we are not targeting, like load, security or usability testing, and identifies them as non-functional testing, but focuses mainly in functionality testing. Within this last one, the methods are presented in relation to the needed understanding of the source code of the web application. When it comes to white box testing, it questions its effectiveness since there are no works developed involving more that just one specific application, considering that it relays heavily on source code analysis which makes it specific for each case. Regarding to other approaches not relying on analysis of the source code, and therefore they are more similar to our approach, two interesting works are presented [22] [23].

In the first of them [23] they focus on addressing the dynamic behavior of web applications, and they use finite state machines (FSM) to model them, which implies that they need to address the state explosion problem that will appear when testing any complex web application. They solve this issue by creating an aggregated FSM from smaller ones that represent parts of the web application, called clusters, and also defining a grammar to specify constraints that limit the possible inputs. Instead, in our approach we use an infinite state machine model, of which we cover a small part in every test execution. The method also has limited automation support and

the user needs to manually define constraints, identify clusters and build the FSMs, and define the test criteria and input selection, which makes the method depending in human input. Like in our case, where the user also needs to define the model, this fact limits the utility of the method for large size web applications. Test sequences can be generated in the form of scripts that can be run using an execution tool. However, the method doesn't address the problem of checking the correctness of the results, something that is done in our case by defining postconditions. QuickCheck also provides features that this method lacks like random test generation, parallel testing and shrinking to small counterexamples.

The method presented in [22] addresses both unit, which relates to single web pages, and functional testing, which addresses complete use cases. To model the web application, it uses an object-oriented model and decision tables instead of FSM. In comparison to the method presented in [23] and our approach, it provides better automation support. However, even if the authors claim the methodology to be pure *black-box*, it makes use of a reverse engineering tool that using the source code of the web application to test, builds a representation model in which the testing is based. This seems to be a poor approach to testing, since they are using the code that they intend to verify to do the verification or testing. It also provides a toolkit that generates test cases, auxiliary drivers to execute them and checks the correctness of the results, needing some help from the user, but much less than in [23]. This makes this approach suitable for large web applications. It is also important to notice that since they use a *ASP.NET* based web application as an example, talk about applying *unit* testing to specific *ASP* files and use automatic reverse engineering we can consider that the applicability of the methodology to different web applications will be very limited by the technologies used to implement them, unlike our approach, which may be suitable to any *RESTful* web application. Also with the use of QuickCheck we need to make a bigger effort to define the test model, but doing so we are able to generate and execute tests that are more specific for the web application under test and therefore are more likely to reveal errors that are difficult to find.

The described analysis [2] also mentions the existence of a big number of commercial testing tools for web applications, but few of them support functional testing, and in such cases they are limited to manage test cases manually created, thus they mention the necessity of a 'greater support to automatic test case generation' . Finally, they also mention as a topic to be addressed to 'provide greater support for dynamic analysis of Web applications'.

As it was mentioned in section 2.3, Wriaki is a *RESTful* web application and the described method can be used with web applications that follow the principles of REST. Regarding to the testing of *RESTful* web applications specifically, a test specification language and a tool for test execution are described in [24], but they provide limited support for functional testing and the test cases need to be specified manually using the specification language. It also supports test validation.

Finally, an approach based on the *Quickcheck* version available for the *Haskell* programming language is presented in [25]. They focus in providing automatic generation and execution of tests cases for *SOAP* web services based on the *Web Services Description Language* (WSDL),

which provides a description of the functionalities that a web service offers. Therefore this method is specific for applications with such characteristics. In addition, considering that the used version of Quickcheck does not support testing with state machines like the one we use, the methodology presented cannot address functionality testing and just provides *unit* testing.

Chapter 6

Conclusions

We have successfully used Quickcheck's state machines to test the functional behavior of a *RESTful* web application, finding one error that lead to inconsistencies in the structure of the data stored in Riak. With the developed model it is possible to simulate a situation where several clients or browsers access the application simultaneously. At the same time, It can also generate and execute complex test cases that test the correct execution of complete use cases providing good testing of the application allowing us to find errors that *unit* testing will not be able to detect. The methodology used can be applied to any other *RESTful* web application applying a similar procedure to the one described, where the main steps are as follows:

- For every resource exposed by the application, which can be identify by an URI, analyze the format of the HTTP requests for each HTTP method supported in that resource
- For each of the previous requests, analyze the format of the response and extract the relevant fields to check its correctness using *postconditions*. When doing this two first steps it is important to consider the possible difference in the execution of the Javascript code between different browsers.
- Use the `eqc_statem` module to define a model where each of the identified requests is a *command* and the *state* is used to store the needed information to know when to perform a request (*preconditions*) and whether the responses are correct or not (*postconditions*).
- To simulate the concurrency, we need to initialize and use several browsers to perform the requests. Each of this browsers will have its own cookie management mechanism and we need to consider and handle this when defining the model. Once this is done, it is possible to define a *property* to have real concurrency or parallelism between the different requests. However, as it was explained in section 4.7, it is important to try to achieve as good parallelism as possible.

It is also important to notice that the fact that Wriaki follows the principles of *REST*, which makes the HTTP messages exchanged with the application stateless and independent of each

other, allowing us to easily generate the requests (commands) in our model. Applying the same procedure to an application where this doesn't happen is possible, but the dependencies between the requests will complicate it. The method described can also be applied to web applications bigger that Wriaki. The fact that the application exposes many web resources is not a problem, it can be easy solved by just adding more commands to our model. However, the limitation of this procedure can be the size of the state used in the model. If we have a web application where a lot of information needs to be stored in order to generate and analyze the HTTP requests and responses our state in the model and the number of transitions can grow up to a point when is not possible to handle them.

As an outcome of the testing performed, we can highlight three interesting points. First, we have been able to identify an error in the application under test and we have fixed it. Second, after finding the mentioned error we have develped our model further, providing a good testing coverage of the web application and executing a great number of tests. However, we haven't been able to find more errors, which shows the robustness of the application. Third, as shown in section 4.7, we have seen that in our case we found necessary a better control over the parallel test cases generation provided by Quickcheck.

Finally, we have also shown that Quickcheck provides a good way of testing any application that uses Riak as its storage system. To perform this testing it is necessary to simulate several clients accessing the data concurrently in the way we do it to test Wriaki. In fact, the error found in the application was related with the way it handles the storage of data in Riak.

Bibliography

[1] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[2] G.A. Di Lucca and A.R. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.

[3] John Hughes. Quickcheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-69608-7.

[4] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1.

[5] S. Vinoski. Wriaki, a webmachine application. *Internet Computing, IEEE*, 16(1):90–94, 2012.

[6] R.T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[7] April 2012. URL http://wiki.basho.com/Riak.html.

[8] April 2012. URL http://www.basho.com.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] April 2012. URL http://code.google.com/apis/protocolbuffers/.

[11] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[12] J. Sheehy and S. Vinoski. Developing restful web services with webmachine. *Internet Computing, IEEE*, 14(2):89–92, 2010.

[13] A. Rodriguez. Restful web services: The basics. *Online article in IBM DeveloperWorks Technical Library*, 2008.

[14] C. Sauer, C. Smith, and T. Benz. Wikicreole:: a common wiki markup. In *Proceedings of the 2007 international symposium on Wikis*, pages 131–142. ACM, 2007.

[15] April 2012. URL http://code.google.com/p/creoleparser/.

[16] April 2012. URL http://github.com/evanmiller/erlydtl.

[17] April 2012. URL http://docs.djangoproject.com/en/dev/topics/templates/.

[18] April 2012. URL http://github.com/basho/wriaki.

[19] 2012 April. URL http://www.charlesproxy.com/.

[20] L.M. Castro and T. Arts. Testing data consistency of data-intensive applications using quickcheck. *Electronic Notes in Theoretical Computer Science*, 271:41–62, 2011.

[21] N. Paladi and T. Arts. Model based testing of data constraints: testing the business logic of a mnesia application with quviq quickcheck. In *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, pages 71–82. ACM, 2009.

[22] G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. De Carlini. Testing web applications. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 310–319. IEEE, 2002.

[23] A.A. Andrews, J. Offutt, and R.T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, 2005.

[24] S.K. Chakrabarti and P. Kumar. Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pages 302–308. Ieee, 2009.

[25] YZ Zhang, W. Fu, and JY Qian. Automatic testing of web services in haskell platform. *Journal of Computational Information Systems*, 6(9):2859–2867, 2010.