Strategies for Testing Client-Server Interactions in Mobile Applications *

How to Move Fast and not Break Things

Niranjan Tulpule
Google Inc.
niranjan@google.com

Abstract

Modern smartphone ecosystems have their unique set of constraints which makes testing the contract between client and servers hard. In this paper we will describe the Google+ team's approaches to solving this problem. We will describe our testing philosophy followed by a couple of frameworks and test design patterns that we have found to be really useful in a client-server testing context.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]

Keywords mobile;protocol testing;

Background

A vast majority of native mobile applications talk to a backend server for various reasons. Some applications such as mobile games only need to communicate with their servers for authentication, downloading new levels of play and storing scores and leaderboards while multiple player games require a constant, active internet connection. Other applications such as social networking applications, local search and discovery apps need to be connected to the internet to function.

The Google+ server and web engineering teams release new versions of services multiple times per week but release the native versions of the iOS and Android applications less frequently, usually twice monthly. This mismatch in the client and server code deployment rhythm makes it much more likely that an engineer on the server team can make a change which breaks the client-server contract between the mobile applications and the server. We often encounter users running old versions of their applications requiring that the server changes have to be backwards compatible for a few versions.

Testing this client-server contract in an automated fashion, which provides the right set of testing safety nets preventing code with defects that break this contract from being committed to the source repository is the only way to sustain the fast development and release cycles on the server

* Experience report based on the work done by the Google+ Mobile Engineering Teams

MobileDeLi '13, October 28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2603-2/13/10. . . $15.00.
http://dx.doi.org/10.1145/2542128.2542134

Testing Philosophy

Testing software at Google's scale is a unique challenge due to the sheer volume of code being committed to its source repository. We routinely see more than 20 changes per minute which require more than 75 million automated test cases to be run each day[1]. The continuous integration system uses dependency analysis to determine all the tests a change transitively affects and then runs only those tests for every change[2]. In spite of these optimizations, a poorly designed test suite can result in wasteful use of resources and more importantly can cause productivity issues for engineering teams.

The Google+ engineering team tries to adhere to the following guiding principles so that we can deliver high quality software at a rapid pace.

- We prefer to prevent code changes with defects being committed to the source repository over finding these defects. Most of our automated test infrastructure on the Google+ team is designed to run tests as a part of the process of submitting a change to the source repository.
- We prefer to keep our test sizes small and encourage all engineers to write unit tests before they can tackle other tests. All our functional tests are optimized for a faster turnaround time which sets in motion a virtuous cycle of developers authoring and sustaining more such tests and helping us achieve higher code coverage.
- We keep all of our test environments "hermetically sealed" [3] which means that our tests should not depend on any ancillary servers and resources which cannot be compiled and deployed on the same test runner.

We will now take a look at two different approaches used to test the client-server interactions for the Google+ mobile applications for the Android and iOS platforms.

Monolithic Test Setup for Client and Server

In our initial iteration we used a monolithic approach to develop our client-server test framework. The test set up process works as follows:

First We build and deploy our server stack on a remote test execution agent using a Hermetic Server configuration similar to the one described by Narla, Salas [3].

Second The test runner processes start a specially crafted version of the Android Emulator on the same remote test execution

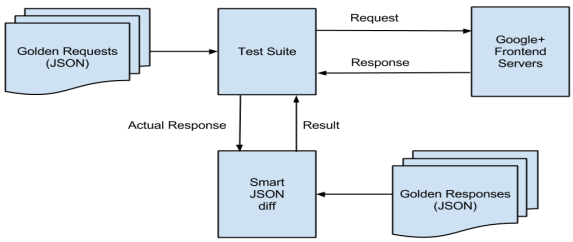

Figure 1. Google+ Mobile Client-Server Test Architecture

agent configured to divert the network traffic by the Application Under Test (the Google+ Android application) to our local server and also enable some options for logging and debugging.

Third The test runner processes then install our Application Under Test on this emulator and then run our suite of functional tests on the application.

This approach made it easy for engineers to author tests, especially for engineers who were new to mobile app development but had a background in developing web services such as Gmail, Google Maps, etc. After running these tests for a few months we realized that they were very costly in practice as they were quite flaky and hard to debug and isolate the cause of failures. They were being used to validate server behavior as well as client functionality.

Through analysis of our test usage and defects, we realized that almost 99% of the changes being made by the developers were either purely server changes or purely client changes and protocol inconsistencies accounted for a vast majority of the defects. We were consuming too much time and resources testing server code when only the client code changed and vice versa.

Testing the Protocol

We decided to try a new approach in which rather than installing every supported version of the application and having to maintain a separate test suite for them, we decided to replace the application by a lightweight stub and use it as a source of requests to the server, which the test suites could then use to verify the correct behavior of the protocol for the given version. Figure 1 shows a simplified view of this test setup. This approach has three major components:

Test Generator or "Record Mode"

We instrumented the Google+ mobile applications on iOS and Android to log the contents of each requests to our server and the corresponding responses. The data contained in requests and responses is serialized using the JavaScript Object Notation (JSON) format. We added a special "Record Mode" option to the versions of these applications used for testing. When we run our application in the "Record Mode" all the requests and responses get stored within the application and can be easily extracted out and used for testing. We call these stored requests and responses as "Golden".

Test Suites or "Replay Mode"

We then launch our Servers under Test using the Hermetic Servers configuration. They are configured to use the same user accounts and data stores, which were used to generate the "Golden" responses. This ensures that there is consistency of test data. We then launch our test suite which uses the JSON data stored in these "Golden Requests" to send request to the Server under Test and when the server responds, it compares the data in those responses to the "Golden" responses using a JSON comparison tool.

Smart JSON diffing

This tool does a smart comparison of the JSON files. There are a few reasons why the actual responses sometimes differ from the stored "Golden Responses" and to account for these nuances we developed a smart JSON diff tool.

First Actual response has an extra field. This is usually harmless as our wire formats are designed to be extended and mobile clients will not fail. If a server developer's change gets flagged then they can easily update the "golden" JSON responses.

Second Value of a field is different in actual and golden responses. There are fields that have dynamic data such as timestamps which are usually harmless for the tests. When we store the golden responses using the "Record Mode" we annotate it to ignore values of those dynamic fields. On the other hand if the field contains a static data and the value has changed it means that this change might break mobile clients if they using this field. This will cause the test suite to fail.

Third Actual response doesnt have a field that is expected by "golden" response. This usually indicates that the mobile clients will break and hence the test suite will fail.

Conclusions

This approach has improved the quality and coverage of our tests dramatically. The running time for tests was cut down from over 30 minutes for every supported version of the application to less than 5 minutes. Over a six month period, the replay tests exhibited less than 1% flakiness over 1000 daily invocations of the tests. These test have prevented many server defects which would have affected the Google+ mobile applications.

Acknowledgments

I would like to thank the engineers who have designed, developed and evangelized the testing solutions described above: Eduardo Bravo, Matthew DeVore, Grygorii Luchytskyi and Sreevidya Tangellamudi.

We owe a huge amount of gratitude to all the talented engineers across Google who have contributed towards building an amazing developer productivity ecosystem.

References

[1] John Micco. Continuous Integration at Google Scale At the EclipseCon 2013, Boston, MA. URL `eclipsecon.org/2013/sites/eclipsecon.org.2013/files/Continuous\%20Integration\%20at\%20Google\%20Scale.pdf`

[2] Pooja Gupta, Mark Ivey and Jon Penix. Testing at the speed and scale of Google On the Google Engineering Tools Blog URL `google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`

[3] Chaitali Narla and Diego Salas. Hermetic Servers On the Google Testing Blog URL `googletesting.blogspot.com/2012/10/hermetic-servers.html`