Abstract

Tango is a free mobile messaging service with 200+ million registered members available on iPhone, iPad, Android phones and tablets. Tango's large user base, broad platform support, frequent releases, and organizational design pose challenges to achieving quality. In this paper I discuss how we are addressing these challenges from our organizational and testing culture, as well as how we utilize test and deployment automation.

We have a culture at Tango of pushing the limits of what is possible and the same applies to our testing mindset. We are constantly adopting tools and techniques that hit at the core of problems with minimum bureaucracy so that we can continue on a release cadence of every 2 weeks.

This paper touches upon how we have approached testing as well as other ways in which we have attempted to address the challenge of shipping with high quality. Some efforts were more successful than others and I will share some lessons learned.

Biography

Amit has over 15 years of experience in software quality assurance as a tester, test architect, test manager, and Director of Technology at Motorola, Veritas, Symantec, Microsoft, and now Tango. He has presented in numerous testing conferences including CAST and Star West.

Amit holds a B.S. in Computer Science from the University of Illinois at Urbana Champaign

Copyright Amit Mathur 2014
1 Introduction

Mobile communications is hot and Tango is at the center of the mobile social communications race. I would describe the race as more of a sprint than a marathon. This has led us at Tango to converge towards a 2-week release cycle.

As if releasing every 2 weeks isn’t challenging enough, we also try hard to give vertical feature teams a lot of autonomy to get their features out which poses interesting challenges when we attempt to integrate all the pieces together.

2 Organizational Design

From a testing perspective our over-arching strategy is to have developers test the code that they write and to provide them with the necessary tools and motivation to do a thorough job.

We are organized into multiple vertical teams that own the key resources that they need to deliver on features. These resources in include Android Developers, iOS Developers, Server Engineers, and Test Engineers.

There are some horizontal teams, whose job it is to support the vertical teams in getting their features out with high quality. The horizontal team are EVIL (core server infrastructure), Server QA (integration test for servers), and Client QA (client end to end testing).

3 Wine Helps Us Achieve Quality

Wine is a big deal at Tango (due to in no small part our VP of Engineering and Chief Wine Geek Gary Chevsky¹). In addition to having wine tastings, we have found ways to utilize wine to help us achieve quality using the construct of a ‘Wine Bet’.

A Wine Bet is placed between a vertical team leader and the VP Engineering wherein a specified dollar value of wine is bet against the number of high priority bugs that are found by the test team after a feature branch is merged to trunk. If greater than the number of agreed upon bugs are found by QA then the vertical team leader loses the bet. The idea is to incentivize teams to only merge to trunk when they are ready. Usually the vertical team lead has to buy a bottle of wine.

¹: Gary Chevsky is an actual person and not a fictional character.
4 Automation

4.1 Overview

At a high level, we have three test frameworks with different goals. The UI automation frameworks are designed to test the UI layer and below (all layers) for end-to-end coverage. The FeatureTest framework is designed to cover the client cross platform layer to server layer. The ServerAPI tests are designed to specifically target servers.

All frameworks are designed so that test cases can be annotated and results sent to a central analytics server to provide a high level view of all tests and the ability to drill down to specific test failures to reduce risk associated with product and test issues.

4.2 UI Automation

Our original UI automation strategy was to leverage Android and iOS UI automation to cover our new client UI during a release. The problem is that our UI changes very frequently (daily) and our automation ends up continuously broken. We decided that instead of spending tons of time during the release fixing the functional tests we would focus our efforts on two fronts:

1. Performance/Reliability – The performance and reliability of our clients are critical as they have a direct impact on the user perception of quality and their engagement. We realized that we simply could not do performance and reliability without automation since it requires running in loops and taking time measurements. Towards the end of the release, we run performance automation for core scenarios as part of the final sign off to ensure there is no performance regression for new clients.

2. Server Regression Validation – Although we get solid targeted server coverage via ServerAPI tests and client/server coverage via the FeatureTests that can be targeted towards server environments, we still like to ensure that when we release new server code that our existing production clients do not break. This is accomplished by continuously running the current production client release as monitors against our server environments to ensure that core users scenarios are working end to end. This does not suffer from the problem of client UI constantly changing since we are keeping clients fixed and changing the servers.
4.3 Feature Test Framework

The Feature Test Framework is a hermetic\(^2\) system (self-contained server system with no network connection) utilized by developers to author end to end tests that can be run on their development machine. The system consists of a test client that is a light wrapper on top of the cross platform client layer that contains the bulk of the client code that gets released with the exception of the user interface layer.

Developers are required to author feature tests to ensure that they have end-to-end coverage. The key benefit of feature tests is that developers can understand the impact of changes throughout the ecosystem on their components. For example if they have a dependency on a downstream component that changed, they may witness their feature tests breaking as a result and quickly get the issue resolved with the relevant party.

Finally, a subset of feature tests (indicated via the dashed line above) can be run against deployed environments. They serve as tests for environment stability and some of them are turned into production server monitors. They serve as good monitors since they simulate user behaviors very well as they contain the key parts of the clients that users actually use.

4.4 ServerAPI Tests

ServerAPI tests are test cases that specifically target direct server interactions. As our services are RESTful (Representational State Transfer), the tests perform http requests and verify the correct http responses are received.

The idea behind having targeted server tests is that they can be developed and executed without the client being present. In particular, it allows us to target servers directly that only interact with other servers.

5 Fault Injection with Mosh

The Tango is a beautiful and elegant dance. When things go right in production it is equally as beautiful. However, we find that things often do go wrong, and the environment looks more like a mosh pit at a heavy metal concert. As our server side load continues to grow, we constantly hit reliability issues that escape to production.

We created 'mosh' for fault injection system to inject bad stuff (network, cpu, memory, io issues) into our servers in our test environments before they happen on their own in production. For network faults (delays and dropped packets) we utilize netem\(^3\), which is integrated into the linux kernel. For CPU, Memory, and IO faults we use a simple tool called stress\(^4\), which allows us to target these failure modes directly.

We have found that this approach helps us target reliability bugs that only manifest on systems where resources are scarce as well as bugs related to the impact of a dependent component being down or slow. In particular, dealing with timeout settings for dependent components. We also use this to see how the component recovers once faults are introduced.
6 Test Cases

We like to keep our test plans as light weight as possible and provide some visualization of areas of coverage. As such we have adopted the ACC\(^5\) (Attribute/Capability/Component) methodology:

**Capabilities:** These are the "verbs" of the system which contain a feature and attribute mapping. (e.g. text messages should be received within 2 seconds of being sent 99% of the time)

**Attribute:** The "adjectives" of the system (Functionality, Performance, Reliability, Usability, Security)

**Feature:** A feature is a "noun" of the system (e.g. video, text, avatar, etc).

**Components:** These are Tango components that implement the capabilities. (e.g. facilitator, authTokenServer, etc). Components can have dependencies on other components.

All automated test cases can be annotated with the above information so that we can visualize risk areas in real time. The key benefit is that instead of seeing a list of failures, we have a better way to know which specific areas are at risk so that we can make quicker informed decisions on how to proceed with the release.
7 Visualizations

7.1 Team Testing Dashboard

We want the vertical teams to 'own' the quality of what they built so we built a Test Testing Leaderboard which helps the entire company visualize where each team is at with their test automation (coverage, what was being run, product issues found, and test issues).

The idea is to gamify testing so that teams feel motivated to execute and keep their test automation clean. This encourages a bit of healthy competition between teams. All test automation is run continuously via our Jenkins continuous integration system and fed in real time to this dashboard.

Once nice element of the dashboard is that it gets updated in real time and provides a stock ticker like up/down symbol providing continuous feedback loop to test teams.
7.2 Component Dependency Visualization

We have over 150 server components that support the backend systems for Tango. It is hard to know when a server component changes what possible impact that could have on the entire system. We built an automated visualization system to help us deal with dependency detection and visualization.

In the example above, if you hover your mouse above a component it will depict all inbound and outbound dependencies on that component. The way that dependencies are detected is based on scanning component configuration information (property files).

This tool has been helpful in both test enumeration and design reviews as it provides a ground truth understanding of dependencies and not what people remember or what exists in outdated design document on the wiki.
8 Change Awareness and Impact Analysis

Given the quick release cycle and largely autonomous feature teams it can be challenging to keep track of what is important when lots of things are changing all the time. To help with this we adopted Phabricator to enable everyone to subscribe to changes in areas of the code that matter to them.

The QA teams use this to subscribe to changes in components that they own or depend on. This allows conversations to happen earlier than they would normally and hence increase the likelihood to the component being certified on time.

9 Unfortunate Escapes

With the engineering systems we have in place we try to catch all bugs before they escape to production but unfortunately we are not perfect. Here are some issues that evaded our systems and wreaked havoc in production.

9.1 Verizon Voice Mail

The way Tango works is that users register with a phone number. If that phone number is already registered by someone else then we provide a mechanism via SMS to verify that number. We also provide the ability to publish the contact list on your phone to our servers so that we can tell you who else is on Tango. This works well in most scenarios because (1) most people use their own phone number to register and (2) most people only have dozens of contacts on their phone.

One day neither of these assumptions held as someone decided to register the Verizon Voice Mail address as their Tango phone number. We allowed that since nobody else had claimed that number. The issue that this caused was the entire system came screeching to a halt as our servers were busy attempting to perform reverse lookup on an entry with millions of users since by default many users have the voice mail number in their contact list.

As a result of this we added new kinds of tests to our system that model this hyper-connectivity. We use this to ensure that our backend systems are designed to deal with highly connected people.
9.2 DoS Attack (a.k.a we found the enemy and it was... us)

Having hundreds of millions of mobile clients hitting our backend services requires constant vigilance to ensure that as load patterns change we are able to adapt to the changing behavior. It is common to see spikes in services due to new features that we roll out.

Recently we released a client that had a bug which hammered our backend services almost causing an outage. The issue was related to feedback logs that are sent from the client to our servers to allow us to know about issues. Unfortunately, in this case the feedback log was an issue in itself.

One step that we have added to our certification process is to sniff the packets from a test client to our backend servers to see if there are any large changes in the number of requests. Unfortunately this methodology is not robust since it isn't always possible to trigger the error condition in the client at the time that the packet sniffing is in effect. The most helpful way we have found to address this risk is to slowly release new clients to production and closely monitor the increase in load.

10 Conclusion

Based on these (and other) test investments we have been able to scale up the company dramatically while steadily improving the quality of our client software and servers. We still have a lot more work to do to ensure that nasty bugs don't escape to production but are happy that we are fully focused on continuous improvement.
References


5. ACC Explained https://code.google.com/p/test-analytics/wiki/AccExplained

6. Phabricator, an opensource, software engineering platform http://phabricator.com