Centralized and distributed testing

Simon Kers
skers@kth.se
School of Technology and Health
KTH Royal Institute of Technology
March 2013

David Wikmans
wikmans@kth.se
School of Technology and Health
KTH Royal Institute of Technology
March 2013

Abstract
Developing and maintaining software is error prone and bugs are hard to locate, this paper reviews some testing approaches used for ensuring the behaviour and reliability of software applications, and how testing differs between distributed and centralized systems. Application operability, uptime and availability are concerns for large systems and services but testing these areas can be tricky. In the software industry testing is not only necessary but common practice and with the emergence of distributed systems and the internet, testing faces new challenges. By reducing dependencies of the distributed parts of the system and purposely subjecting the network to high load during fuzz testing, the distributed system can be hardened and better protected against bugs.

1 Introduction
As software projects grow, so does the amount of possible bugs. Software testing has been a part of software-development for as long as there have been computers, and bugs in the shape of logical error, faulty implementation and in modular interoperability. To assess the quality of computer software testing is very important. Testing typically takes upwards 40% – 50% of the development time, and even more for systems that require higher levels of reliability. [3]

Software testing is divided into two schools of thought. Static testing which analyzes the source code by inspections, peer reviews and *walkthroughs*. Dynamic testing where the application is executed and the behaviour of the program, or rather, the systems response is tested. This short paper discusses various methods and surrounding paradigms respective properties, in relation to testing. Testing can be done after the development process by an independent group of testers[1], which is part of the traditional or *Waterfall* approach, other software development processes such as Agile where the developers work with short development cycles, continuously test the software. Related to Agile processes is *Test Driven Development*, where you write tests before the development cycle starts.[3]

This paper focuses on testing strategies, properties of centralized and distributed systems, how they relate to testing and ensuring fault-tolerant operation. Distributed testing requires additional precautions over its centralized parts and they both share some fundamental practices.

[1] As part of verification and validation

2 Evaluation
2.1 Fuzz testing
In order to find unexpected bugs in software, a method called fuzz testing can be used. This is an automated process that supply the system with random input, trying to cause bugs and crashes. The method can find bugs that human beings could easily overlook, enhance integrity and is generally easy to implement.[4]

Used in both centralized and distributed software testing, *Fuzz testing* is a form of *black box testing*, useful for locating hard to find software bugs. In section 2.3.2, we take a closer look at the *Chaos Monkey*, which uses fuzz testing to finding bugs in distributed fault-tolerant systems running on Amazons cloud computing platform.

2.2 Centralized systems
2.2.1 Testing methods
In the *black box testing* approach, the tester do not care about the inner workings of the implementation, only its return value. The tester injects test cases through one or more available interfaces and compares to the expected result.[4]

The *white box* or *clear box* approach is when you test the internal structure of the object with knowledge of its inner workings. The *gray box* testing is a combination of both.

Another is the *bottom-up top-down approach*, where bottom-up means writing tests for each component and top-down only the functions of the high level objects.

2.2.2 Unit testing
In *Unit testing* the developer test small units of code, this does not ensure the program as a whole works as intended and has to be combined with application-level testing. Normally these tests are on a functional level. In object-oriented languages unit tests means tests for classes.

2.3 Distributed systems
A whole set of new problems arises when developing distributed systems, the proneness to error that occurs in centralized software are still there in distributed systems. Just as unit testing in regular programs benefit from having distinct modular parts, the distributed application can benefit from the lower coupling. This effectively decreases the amount of software testing specific to distribution, thanks to the reduced inter-node dependencies. This isolates the test cases to a local level and avoids some aspects of distributed testing procedures and simplifies code by treating the nodes as resources.

Distributed system often grow large and produce detailed logs of programmatic operation and network status, analyzing large quantities of log data becomes more important as these systems grow and becomes more prevalent. Though tests serve an important purpose, they are only as good as their implementation and can not guarantee a fault-less system and testing distributed aspects will become an important part of developing large scale systems.

2.3.1 States

Distributed applications, effectively increases the number of possible states the system can have. One meassure against this is employing designs which reduce dependencies between nodes, that the state of one node affects the state of another as little as possible.

Distributed systems rely on communication between nodes, this could lead to an increased number of states the system collectively can take, increasing the possibility of bugs.[1]

2.3.2 Chaos monkey

When testing distributed systems a new class of software bugs emerge which conventional testing procedures may not catch. One inconvenient property of such bugs are infrequent occurrences and are difficult to reproduce, especially with code not in production[2].

One counterintuitive approach would be Netflix *Chaos Monkey* implementation, a disaster testing system, which operates on Amazons cloud computing platform. It randomly causes failures of the virtual machine instances, know as *Auto Scaling Groups*.[2]

This puts the distributed systems resiliency at test and enforces security measure against netsplits, erratic behavior and failure caused by high latency and bugs of such distributed nature. It enables software developers to discover rarely occurring bugs and resolve them before the code goes live.

3 Conclusions

External occurrences such as power outages and network failure are hard to predict, therefore new ways of simulating failure and load are aspects that are more specific to distributing an application are valuable. *Dynamic testing*[3] becomes more important to ensure reliability of the additional aspects such as latency and bandwidth of the distributed system, low coupling and high cohesion still serve as a safeguard against these problems.

[2]Code that is currently in development

[3]Including *black box testing*

4 References

[1] *Erlang '11: Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, New York, NY, USA, 2011. ACM. 553112.

[2] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`, 2008. [Online; accessed 19-July-2008].

[3] Lu Luo. Software testing techniques. `http://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf`.

[4] A. Takanen, J.D. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House information security and privacy series. Artech House, 2008.