

Copyright
by
Iyad Shaher Azrai
2012

**The Report Committee for Iyad Shaher Azrai
Certifies that this is the approved version of the following report:**

**Software Test Automation: A Design and Tool Selection Approach for a
Heterogeneous Environment**

**APPROVED BY
SUPERVISING COMMITTEE:**

Supervisor:

Adnan Aziz

Sarfraz Khurshid

**Software Test Automation: A Design and Tool Selection Approach for a
Heterogeneous Environment**

by

Iyad Shaher Azrai, B.S.Comp.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2012

Acknowledgements

I would like to thank my professors Dr. Adnan Aziz and Dr. Sarfraz Khurshid for their unwavering support and their infinite patience. I would also like to thank my wife Tara for standing by me throughout this journey.

Abstract

Software Test Automation: A Design and Tool Selection Approach for a Heterogeneous Environment

Iyad Shaher Azrai, MSE

The University of Texas at Austin, 2012

Supervisor: Adnan Aziz

This report describes a design approach for implementing a software test automation solution that can accommodate existing test processes in an organization. The process of implementing a software test automation solution is a large undertaking and requires careful planning to avoid unsuccessful implementations. This report outlines a design that can integrate with existing business and development processes in an organization, and recommends automation and development frameworks for achieving the test automation goals.

Considerations for a heterogeneous test environment with varying types of supported operating systems, such as Windows and Linux, and multiple test execution environments, such as Java and .NET, have been made in this design and in the tool selections for the system implementation. The report also describes some of the challenges and caveats of automation in a heterogeneous environment along with recommended solutions to these challenges.

Table of Contents

List of Tables	viii
List of Figures	ix
Chapter 1: Introduction	1
Problem Description	1
Report Organization	2
Chapter 2: Goals and Requirements	3
Standalone Regression Testing	3
Product Install and Configuration	4
Test Harness Install and Configuration	4
Test Execution	5
Results Reporting	5
Cleanup	5
Product Removal	6
Client-Server Regression Testing	6
Product Install and Configuration	7
Test Install and Configuration	7
Test Execution	8
Result Reporting	8
Cleanup	8
Product Removal	8
Performance, Load and Endurance Testing	9
Environmental Constraints	9
Business Constraints	11
Usability Requirements	12
Test Automation System and Environment Assumptions	12

Chapter 3: High Level Architecture and Design.....	13
Chapter 4: Tool Evaluation and Selection	19
Remote Execution and File Transfer	19
Web MVC Framework	24
Chapter 5: Implementation Results, Considerations and Caveats	26
Implementation Results	26
Implementation Considerations and Caveats	26
STAF Automation Lessons Learned.....	29
Chapter 6: Conclusions	31
Future Work	32
Bibliography	33

List of Tables

Table 1: Supported Operating Systems.....	10
Table 2: Test application environments.....	11

List of Figures

Figure 1: Standalone Regression Test Process Overview.....	4
Figure 2: Client-Server Test Process Overview.....	7
Figure 3: Software Test Automation System High Level Architecture.....	18
Figure 4: Sample Python code using STAF.....	22

Chapter 1: Introduction

PROBLEM DESCRIPTION

Implementing a software test automation solution in an organization with established development and test procedures is no small undertaking [1]. Automation projects tend to consume large amounts of resources and time to develop. The benefits of successfully completing a test automation process are also typically slow to materialize due to the heavy initial costs of implementation [2]. Therefore, careful planning is required to avoid common pitfalls in software test automation and to produce a solution that meets the current and future test needs of an organization.

Planning for a test automation effort is especially important in heterogeneous environments. A heterogeneous environment contains a mix of operating systems and software platforms. A heterogeneous environment typically contains tests systems running various operating systems from several major vendors such as Microsoft, Apple, Red Hat, Suse and Ubuntu. Moreover, each of these vendors has a variety of past and present editions of their operating systems that are still under active support [3, 4, 5].

A heterogeneous environment also consists of supporting several software environments for test execution such as JAVA, C/C++, .NET, Python and various other software environments, each with its own constraints and caveats.

For a software test automation solution to be successful in an established development and test organization with a large number variables and permutations in the environment, it would need to integrate seamlessly with the existing processes of the organization and be able to leverage existing test automation tools and frameworks to facilitate supporting the various environmental permutations and reduce the cost of the initial investment in the system.

This report describes the process of designing and implementing a test automation solution that can accommodate a heterogeneous test environment. This was a team effort conducted by a Quality Engineering group in which I contributed to the design of the solution, the tool evaluation efforts and the development of the system. This Quality Engineering group is part of a commercial organization with several software product offerings. We leveraged open source tools and frameworks to implement this software test automation system that transformed testing on different operating systems from a manual process on a handful of platforms to an automated one that covers over thirty different operating system configurations.

REPORT ORGANIZATION

This report describes the goals and requirements for the software test automation system in Chapter 2 with special emphasis on leveraging existing test processes to promote adoption among Quality and Software Engineers. The report describes the architecture and design of the system in Chapter 3 based on the goals and requirements of the group. The report provides tool and framework recommendations based on research and experimentation in test automation approaches in Chapter 4. Implementation considerations and caveats are described with some solutions to the challenges faced using the recommended tools and frameworks in Chapter 5.

Chapter 2: Goals and Requirements

The main objective of this report is to describe the design of a software test automation system that is easy to maintain and can accommodate running existing software test harnesses [6] and processes. The system should be able to support a heterogeneous operating systems environment and test applications from a variety of software environments such as Java, C/C++, VisualBasic, C#, Delphi and several scripting languages Perl, Python and TCL.

The system should be easy to deploy, use and maintain to promote adoption among Quality Engineers and Developers as well. The system shouldn't disrupt processes that are already in place for Configuration Management and Development; it should seamlessly integrate with these existing processes.

This system will address automating the following test patterns: Standalone Regression Testing, Client-Server Regression Testing, Performance Testing, Load Testing and Endurance Testing. There are some common aspects to all these test patterns, but they do offer enough differences that can pose challenges for automation. Describing their use cases should help illustrate the needs of the test automation infrastructure. Note that these test patterns are the internal test processes for the Quality Engineering group that developed the solution described in this report. However, these patterns are most likely common to other quality organizations.

STANDALONE REGRESSION TESTING

Regression testing is the process of running established and repeatable test cases and harnesses on a system to detect software defects, also known as regressions, introduced into a product through the development of new features or through software maintenance. Standalone regression testing is the process of executing these regression

tests locally on a System under Test, referred to throughout the document as SUT. Test applications and the product to be tested will reside on the same system and are executed locally on that machine. The process for conducting standalone tests on a system was designed by the Quality Engineering group for this test automation system and consists the following phases:

Standalone Regression Process					
Phase	Product Install and Configuration	Test Install and Configuration	Test Execution	Results Reporting	Cleanup
	<ul style="list-style-type: none"> • Product binaries copied to SUT • Product installed on SUT • Product configured for testing 	<ul style="list-style-type: none"> • Test binaries copied to SUT • Test configured for execution • Product prepared for testing 	<ul style="list-style-type: none"> • Diagnostics and monitoring started • Tests executed 	<ul style="list-style-type: none"> • Results copied to results repository • Diagnostic test artifacts copied to results repository 	<ul style="list-style-type: none"> • Product configuration restored to post install state. • Test configuration rolled back. • Test binaries, results and artifacts removed from SUT

Figure 1: Standalone Regression Test Process Overview

Product Install and Configuration

In this phase the product binaries or the product install package are copied to the SUT. Then the product is installed and configured for testing. Product configuration changes in this phase tend to be common changes that can be utilized by all the tests that will be conducted on the SUT. Product logging levels could be raised to verbose and other system configurations can be performed such as enabling application debuggers to catch and trap fatal product exceptions.

Test Harness Install and Configuration

In this phase, the test harness is copied to the SUT and installed. Installing a test harness could involve several activities, such as registering libraries, installing supporting products such as an interpreter or a system dependency such as a .NET. The harness can also be configured by performing updates to test configuration files, such as INI files or

PROPERTIES files. Test data preparation is another common task that is performed in this phase. This can consist of creating database tables and populating them with data or simply copying existing data sets to appropriate locations for testing.

Test Execution

Once the harness has been configured, the tests can be executed on this system. In this phase, the test automation infrastructure would need to verify that the test was started successfully and that it completes. A test execution can abort prematurely if the product exhibits a defect and fatally terminates. This could cause the test application to pause or terminate itself. An automation framework would need to be able to detect such scenarios and act appropriately by terminating the test if it is in an inconsistent state and by preserving diagnostic artifacts such as test logs, product logs and core dumps if the product crashed.

Results Reporting

After test execution is complete, or if the test or product failed and diagnostic artifacts were generated, gathering the results is the next step in order to perform result analysis and reporting. Gathering results simply means collecting all result and diagnostic artifacts and copying them to a centralized results location. This is done to avoid having Quality Engineers visit every SUT to analyze test results.

Cleanup

Once the results are gathered, the SUT should be cleaned up in order to queue up other test harnesses for execution and have them start from a consistent system state. Cleanup usually means changing any product configurations that were performed by the test harness and also removing any test data and binaries from the system. Restoring the

product to a state identical to that post the first phase of automation “Product Install and Configuration” is ideal and desired.

Product Removal

This is the last phase in automated testing. Once all the tests have been conducted, the product should be removed from the system to get the system back to a clean state, ready to start the complete Standalone Regression process once again with a newer version of the product as soon as it is available. This can be achieved in several ways: by automating the process of removing the product and removing all artifacts that were installed or created by all the test harnesses that were invoked on the SUT, or by automating the operating system deployment on that SUT, i.e. the operating system is either freshly reinstalled on the SUT or a backup of the installed operating system is restored on the SUT.

CLIENT-SERVER REGRESSION TESTING

Client-Server Regression testing has similar phases to Standalone Regression testing, but it involves two SUTs that need to synchronize their progress along these phases. Client-Server Regression harnesses can be variants of Standalone test harnesses that simply exercise the same set of tests over a network to a remote server to verify that networking doesn't introduce adverse behaviors in the product. They can also be specialized regression tests that emphasize network communication by employing a variety of connection methods and networking configurations. The process for conducting Client-Server Regression tests was designed by the Quality Engineering group for this test automation system. Figure 2 illustrates this design and the timelines for the client and server in the Client-Server pattern:

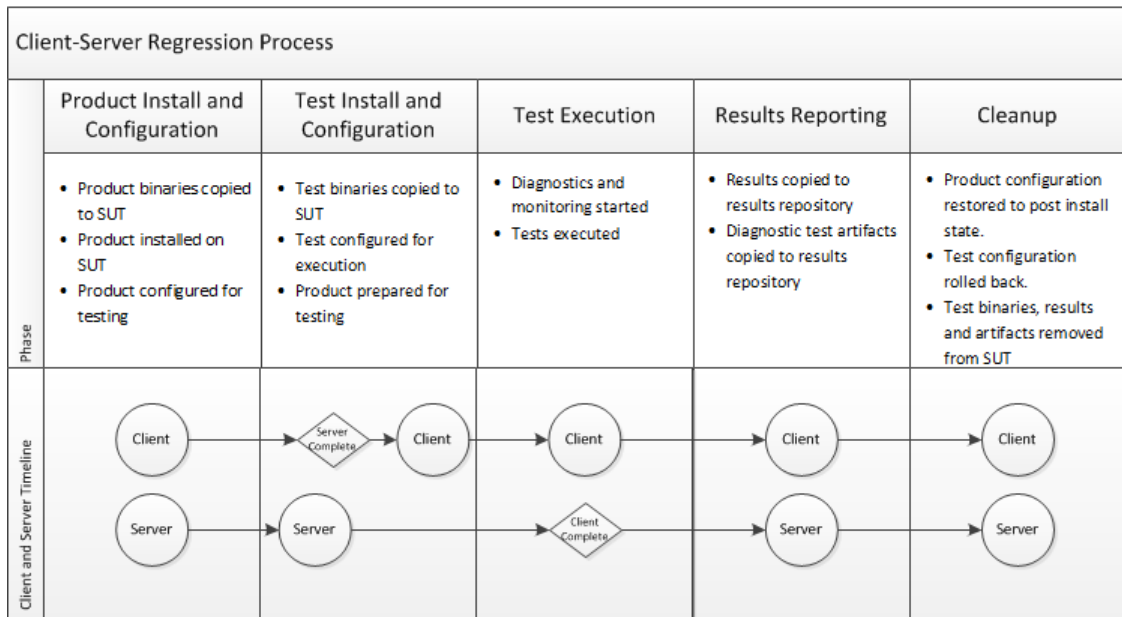


Figure 2: Client-Server Test Process Overview

Product Install and Configuration

This phase is identical to that of the Standalone Regression tests and can be conducted simultaneously on both the Client and the Server SUTs. The product binaries or installer can be copied simultaneously to both the Client and the Server. The install can also be conducted simultaneously by the Client and Server along with any product configuration required for general testing.

Test Install and Configuration

This phase will require synchronization between the Client and the Server as the client typically relies on having certain server elements in place, such as shares to data should be created, specific services started and test data configured and made accessible remotely. There are however certain steps that can be executed simultaneously in this phase, such as copying test harness binaries to both the client and the server. Once the

copies are completed, test harness configuration should be conducted on the server first followed by the client.

Test Execution

Here the test is executed on the client. No actions will need to be conducted on the server as all the test preparation should have already been conducted in the previous phase “Test Install and Configuration”. The server in this phase simply awaits the completion of tests conducted by the client.

Result Reporting

After the tests are complete, copying the test results to a centralized repository is necessary along with any other artifacts including failure artifacts such as core dumps or diagnostic logs. These steps should be executed by both the client and the server and can be done simultaneously. Results from the client and any artifacts from the server should ideally be copied to a common location to form a single test results location. This will help avoid confusion during results analysis.

Cleanup

This phase can also be executed on both the client and server simultaneously. It should leave both the client and the server in a state that is ready for executing another set of Client-Server Regression tests.

Product Removal

This phase is identical to that of the Standalone Regression testing and can be conducted simultaneously on clients and servers. This should be conducted once all the client-server test harnesses are complete to prepare the SUTs for a new set of regression

tests. This will ensure that future tests will be conducted on a system in a clean and consistent state.

PERFORMANCE, LOAD AND ENDURANCE TESTING

Performance, Load and Endurance testing are test activities that require synchronization among several SUTs. They are typically Multi-Client-Server tests, where multiple clients are conducting tests on a shared server simultaneously. Performance tests are designed to determine maximum product thresholds, such as the maximum transactions per second throughput for a database system. Load tests are designed to verify that overcommitting a product's resources does not adversely affect the product. Endurance tests are designed to exercise a product in a typical usage scenario over an extended period of time to simulate extended customer usage of a product.

These three test patterns can be considered variations of Client-Server Regression tests. They simply substitute a single client with multiple clients and follow the same timeline as the Client-Server Regression tests would follow. During the "Test Install and Configuration" phase, all the clients wait on the server to complete its configuration steps. The server also waits on all the clients to complete their test execution before executing the results collection steps. The synchronization solution used for a Client-Server Regression pattern should meet the synchronization needs of these Multi-Client-Server Test patterns.

ENVIRONMENTAL CONSTRAINTS

The software test automation system will need to support an environment that consists of many different operating systems. The operating systems we intend to support are outlined in the table below.

Operating System [3, 4, 5]	Processor Architectures	Supported Editions
Microsoft Windows XP	x86, x86_64	Professional
Microsoft Windows 2003 Server	x86, x86_64	Standard, Enterprise
Microsoft Windows Vista	x86, x86_64	Professional, Enterprise
Microsoft Windows 2008 Server	x86, x86_64	Standard, Enterprise
Microsoft Windows 7	x86, x86_64	Professional, Enterprise
Microsoft Windows 2008 R2 Server	x86_64	Standard, Enterprise
Microsoft Windows 8	x86, x86_64	Professional, Enterprise
Microsoft Windows 2012 Server	x86_64	Standard, Enterprise
Red Hat Enterprise Linux 4	x86, x86_64	Desktop, Server
Red Hat Enterprise Linux 5	x86, x86_64	Desktop, Server
Red Hat Enterprise Linux 6	x86, x86_64	Desktop, Server
Suse Linux Enterprise 9	x86, x86_64	Desktop, Server
Suse Linux Enterprise 10	x86, x86_64	Desktop, Server
Suse Linux Enterprise 11	x86, x86_64	Desktop, Server
Ubuntu 10.04 LTS	x86, x86_64	Desktop, Server
Ubuntu 12.04 LTS	x86, x86_64	Desktop, Server
Mac OS X	x86_64	10.6, 10.7, 10.8

Table 1: Supported Operating Systems

There are also several software application environments that will need to be supported for testing purposes. Test applications can be developed in any one of the following application environments:

Application Environment	Typical Applications	Console/Graphical Interface
JAVA	JDBC application	Both
C/C++	ODBC application	Both
C# .NET	ADO.NET application	Both
Visual Basic	OleDb or ActiveX	Graphical
Perl	Test automation	Console
Python	Test automation	Console
TCL	Test automation	Console

Table 2: Test application environments

The expectation here is that the test automation system would not be tightly coupled with any one of these application environments; it should merely be a conduit for executing these applications. The test applications should not have any dependencies on the automation system; an Engineer should be able to continue executing the test applications without the presence of the automation infrastructure. A dependency on the new automation system should not be introduced in any of the existing test applications or harnesses.

BUSINESS CONSTRAINTS

Another constraint on this effort is integrating an automated software test solution into an already well-established development and configuration management environment. The constraint here is to not disrupt the existing processes and ideally not force any changes or dependencies on them.

Developers and Quality Engineers should be able to continue maintaining their test environments independently from maintaining the automation infrastructure. They should be able to continue executing their test harnesses and applications without deploying the new test automation system.

No changes to configuration management are expected as well other than the introduction of a new project for the software test automation system. The expectation is that the software test automation system will support the existing configuration management process by detecting when new software builds are available and by obtaining these builds when necessary.

USABILITY REQUIREMENTS

Usability requirements for this product are geared towards maximizing the adoption of this new system by the Quality Engineers and even Developers. The solution should be easy to deploy, preferably a “Turnkey” solution with everything included. Detection of SUTs should be done dynamically; i.e. the operating system and configuration of the SUT should be deduced rather than provided to the system. This will simplify the process of configuring a freshly deployed software test automation system.

TEST AUTOMATION SYSTEM AND ENVIRONMENT ASSUMPTIONS

The following is assumed to be true for the test environment and the test automation system:

- A system under test can only have one product installed at a time.
- A system under test can only have one test running at a time. A test can utilize more than one running application simultaneously, but no two different test harnesses can be executed simultaneously to avoid configuration, execution and cleanup conflicts.

Chapter 3: High Level Architecture and Design

The process of architecting this system needs to start with defining its environment and identifying what components would be required to meet the goals and requirements of the system. The environment as we described earlier is a heterogeneous environment that supports multiple operating systems and application platforms. The environment is networked as it is expected to support client-server tests along with performance, load and endurance tests that utilize multiple clients simultaneously connected to a shared server. We can conclude from this that the environment is a basic computer network consisting of several computers connected to a single or multiple network switches. We can assume that there is infrastructure in place for managing network addressing such as a DNS and DHCP server.

The system will need a cross-platform method for invoking remote commands on various operating systems, and a cross-platform component for transferring files to and from various operating systems. These components will need to reside on all the SUTs and will most likely involve introducing new technologies to the test environment.

A master-slave architecture pattern [7] will be used for developing this system. The SUTs will be considered the slaves in this architecture; a master server is expected to delegate tasks for them to complete. A new system will need to be introduced to the environment to act as the master server and will be referred to as the Test Automation Server, TAS for short. This server will have to perform several functions: stage the product and test binaries, copy the product and test binaries to the systems under test, house and delegate test configuration and execution steps on remote systems, gather test results and artifacts from SUTs and store them in a results repository.

We have identified two required repositories for the system so far: one for staging which can simply be a file system directory location, and a results repository which will most likely also be a file system directory location to store all the test results and artifacts. The file transfer and remote invocation components implemented for the SUTs will be reused on the TAS to provide the functionality for transferring files from and to the SUTs and the staging and results repositories. The remote invocation component will be used to execute the configuration steps and the software tests on the SUTs.

An additional component will be required for housing common utilities that will be deployed on the SUTs. This is necessary to maximize the delegation ability of the TAS. This component will contain a wide range of utilities such as tools for performing compressed archive management and system configuration scripts. This element of the design should help minimize the required features of the remote invocation component.

The design of the system will largely depend on the tools and frameworks we select for implementation. There are however some high level design items that can be elaborated to help with the tool and framework selections. The first component to consider is the common utility repository. This component will need to be able to perform the following tasks:

- Recursively traverse test file system directories and perform token substitutions on test files such as INI or PROPERTIES configuration files.
- Create and delete files and directories.
- Change file and folder permissions and ownership.
- Manage the compression and deflation of file archives.

As for the TAS design, a simple Model-View-Controller (MVC) [8] application will meet the desired functionality from the server. Most modern web frameworks can generate good scaffolding for the Controllers and Views and will not need many

alterations to provide basic functionality with room to grow and improve in the future. This design will focus on the model of the system as it will help describe the expected behavior of the system.

The model follows the physical environment very closely. The entities that are of interest are the SUTs, software products and the test harnesses. I will start by describing the important attributes of the SUTs.

To successfully conduct a test on a system, we would need to know the following attributes of the system:

- System name or IP Address (name or address needed for communication)
- Operating system
- Operating system family: Windows, Linux, Macintosh
- Processor Architecture: x86, x86_64
- System state: Clean, product installed, running tests.
- Supported software environments: Java, .NET, Python, etc.

The operating system, and specifically the operating system family, will determine the syntax required for the remote execution calls to that system from the remote invocation component. The combination of the operating system and processor architecture will also determine what types of products are supported on that SUT. That information along with the supported software environments will determine which tests can be executed on the SUT.

The supported software environments can also be used to alert the tester of possible misconfigurations on the system such as not having the Java Runtime Environment installed.

The state of the system is necessary to enforce repeatability of the tests. You wouldn't want to conduct a test on a system that is already conducting a test or that has

failed to clean up after the completion of a test. The System state attribute will be used to enforce these testing practices.

As for the Test model, we would need to know the following attributes:

- Test or test harness name.
- Test pattern: Standalone, client-server, performance, etc.
- Supported products.
- Supported operating system families.
- Supported processor architectures.
- Required software environment: C, Java, .Net, Python, etc.
- Test procedure.

The supported operating system, processor architectures and software execution environment and product fields are necessary to match up the test harness with compatible SUT candidates. The Test model has a many-to-many relationship with the SUT model.

The procedure attribute specifies the sequence of steps for the test harness and test pattern. It will follow the expected test pattern steps with specific actions for configuring and executing the test harness. This attribute will simply hold the name of a test script that is provided by the common utilities component. This utility is expected to be executed on the SUT through the remote invocation component. The test procedure is specified in a script that is copied over to the SUT versus a procedure that resides on the TAS to adhere to the master-slave architecture pattern by facilitating the delegation of the test configuration and execution. Having a script that can be copied to the SUT would also simplify manual troubleshooting as the script execution can be isolated from the TAS remote invocation mechanism.

The Product model is a simple one. The following product attributes are required for building the system:

- Product name.
- Supported operating system family.
- Supported processor architecture.
- Product install procedure.
- Product removal procedure.

The product supported operating system families and processor architectures are used to determine which systems under test are compatible with the product. The product install and removal procedure attributes are similar to the Test model's test procedure attribute in that they point to a utility from the common utility component to be executed on the SUT when the product is to be installed or removed.

The Product model has a one-to-many relationship with the SUT model. This is based on the assumption stated in the Goals and Requirements section that no more than one product can be installed on a SUT at any given time. The Product model has a many-to-many relationship with the Test model.

Figure 3 illustrates the complete high level architecture of the system:

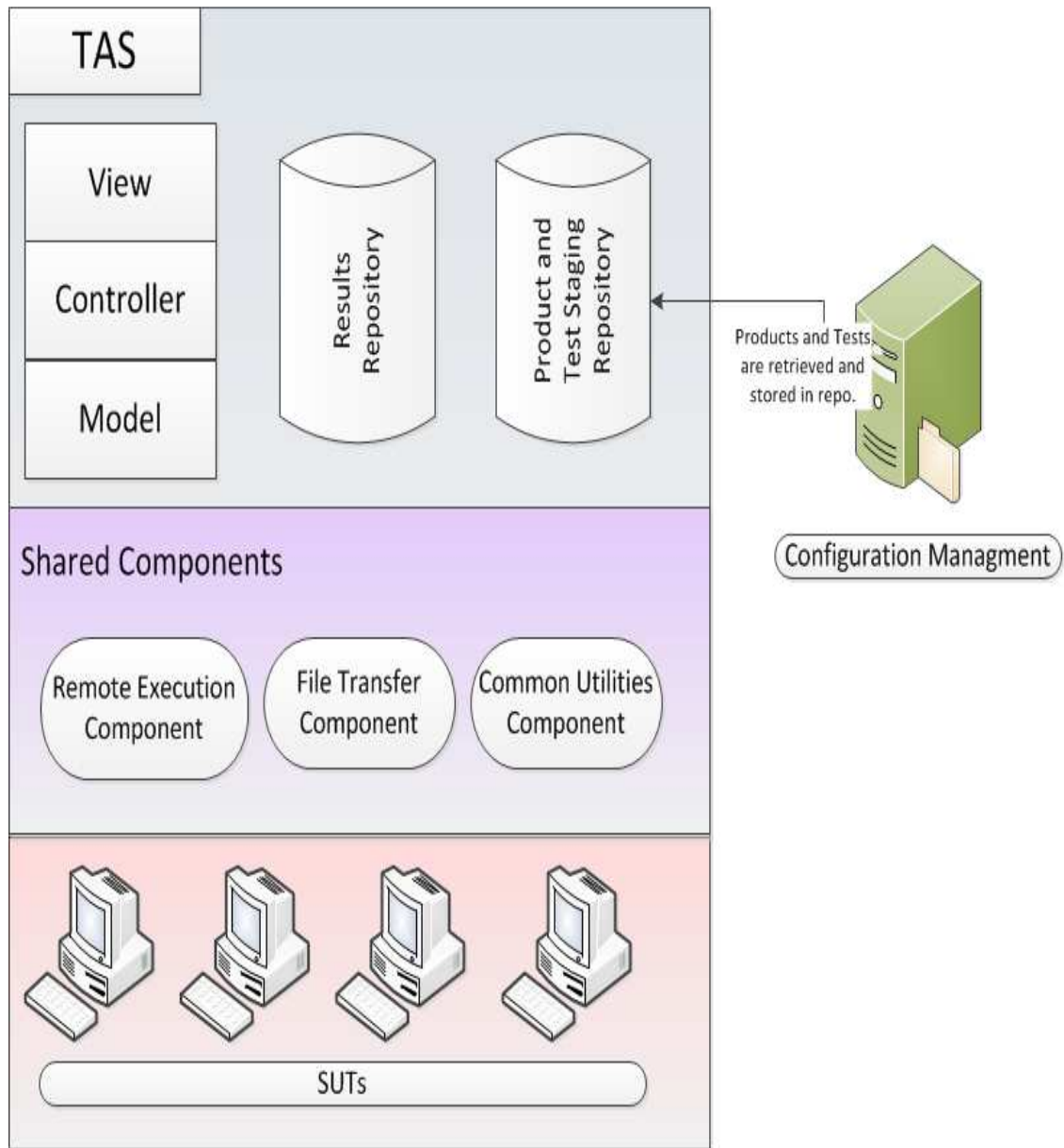


Figure 3: Software Test Automation System High Level Architecture

Chapter 4: Tool Evaluation and Selection

The evaluation process for finding tools and frameworks to implement the software test automation system focused on finding open source and well maintained tools that can be drop-in, low configuration, easy to maintain and use components. We can list the tools and frameworks required for implementing the test automation system as follows:

- A tool or component to handle delegating tasks and invoking commands on multiple remote operating systems.
- A tool or component to handle file transfers between multiple operating systems.
- A tool or component to query remote systems for configuration properties. This is necessary for dynamically determining the operating systems and other attributes of the systems.
- A tool or component that can synchronize actions between multiple systems.
- A programming environment that is both suitable for developing components for the TAS infrastructure and the common utilities component.
- A web framework to build the TAS MVC application.

REMOTE EXECUTION AND FILE TRANSFER

Shell based utilities were first examined to accomplish the task of transferring files and invoking remote commands on SUTs. Tools such as SSH [9] and RSYNC [10] presented problems due to cross-platform compatibility issues and the need to store login credentials or certificates per SUT. Setting up an environment that can utilize SSH and RSYNC for test automation would have required additional infrastructure changes, such as configuring SAMBA [11] and Windows shares for file transfers. The attempt to use

these tools was abandoned for a more robust and cross platform solution that was designed for test automation.

STAF [12], Software Test Automation Framework, is a tool that was originally developed by IBM for test automation and was subsequently released as an open source project [13]. STAF is a multi-platform and multi-programming language framework for test automation. It provides many reusable services to facilitate the process of test automation such as file transfer, remote execution and querying remote systems for configuration properties. STAF was originally released under the GNU Lesser General Public License V2.1, then under the Common Public License V1.0 after the STAF V2.6.8 release, and is now distributed under the Eclipse Public License V1.0 and has been since the STAF v3.2.5 release. STAF is at version v3.5.4 as of this writing. Several projects have successfully used STAF to automate their testing activities; Cervantes [14] from the Jet Propulsion Laboratory describes the experience of successfully implementing an automated test framework using STAF and selecting it over building an in-house solution or using proprietary solutions that were recommended by consultants. The project is still under active development and has a vibrant community.

STAF requires a client to be installed on a system in order to perform its tasks. Installing the client is a simple process. Both graphical wizard installers and simple compressed archives are provided for all the supported operating systems and processor architectures.

STAF provides many services out the box. The ones of interest to this effort are:

- The Ping Service. This service is used to determine if a STAF client is running.
- The Variable Service. This service is used to query a client for system and shared variables. System variables can be internal STAF configuration

variables or system environmental variables. Shared variables are dynamic variables that can be used by test applications.

- The File System Service. This service allows you to interface with the file system on STAF clients. It can be used to transfer files from and to a system.
- The Process Service. This service allows you to start processes and execute commands on a STAF client.
- The Semaphore Service. This service can be used to synchronize multiple client activities and access to resources. This service is useful for synchronizing clients in performance, load and endurance testing.

STAF is a multi-language framework. It can be accessed through C/C++, Java, Python, Perl, TCL and ANT. The language selection for implementing the system will largely depend on the web framework that will be used to implement the TAS MVC application. A Python framework was ultimately selected for this project, and is described later in this document, so the Python access method was used for developing the automation solution around STAF. Here is a sample Python application that illustrates the usage of the main STAF services that are of interest to this project.

```

1 from PySTAF import *
2 import sys
3
4 try:
5     handle = STAFHandle("Test")
6 except STAFException, e:
7     print "Error registering with STAF, RC: %d" % e.rc
8     sys.exit(e.rc)
9
10 result = handle.submit('remote', 'ping', 'ping')
11
12 if (result.rc != 0):
13     print "Error submitting request, RC: %d, Result: %s" \
14           % (result.rc, result.result)
15
16 result = handle.submit('remote', 'var',
17                        'resolve string {STAF/Config/OS/Name}')
18
19 if (result.rc != 0):
20     print "Error submitting request, RC: %d, Result: %s" \
21           % (result.rc, result.result)
22 else:
23     print "OS Name: %s" % result.result
24
25 result = handle.submit ('local', 'fs',
26                        'copy file "/stage/install.exe" tofile \
27                        "/stage/install.exe" tomachine remote')
28
29 if (result.rc != 0):
30     print "Error submitting request, RC: %d, Result: %s" \
31           % (result.rc, result.result)
32
33 result = handle.submit ('remote', 'process',
34                        'start command "/temp/install.exe" wait stderrtostdout returnstdout')
35
36 if (result.rc != 0):
37     print "Error submitting request, RC: %d, Result: %s" \
38           % (result.rc, result.result)
39 else:
40     print "Result: %s" % result.result
41
42 rc = handle.unregister()
43 sys.exit(rc)

```

Figure 4: Sample Python code using STAF.

A handle needs to be acquired in order to process STAF requests as illustrated on line 5 in Figure 4. The local STAF client needs to be running in order to obtain the handle otherwise an error and return code of 21 is returned to the caller. Once a handle is acquired, STAF requests can be submitted. STAF requests always take the host name or IP address of the target system as the first parameter, followed by the desired service to be used. Then the specific service request is passed and is processed by the STAF engine that returns a STAFResult object consisting of the return code and the result string.

Line 10 shows a simple ping service request to a remote host called “remote”. This request simply validates that the STAF client is running on the remote host. Line 16 uses the Variable Service (VAR) to retrieve the remote host operating system name. An abbreviated form of the operating system name is returned such as Win7 or WinSrv2008. Line 25 shows an example of using the File System Service (FS) to copy a file from the local system to the remote system. Note that in this example the remote host is a Windows system, but STAF can handle Unix-style file system paths on Windows which simplifies supporting the various operating systems required for this project. Line 33 shows an example for the Process Service which is used to invoke commands and applications on the remote system. Note that this example uses some decorators to control the behavior of the Process Service call return. The “wait” decorator signifies that the call should only return once the remote process completes and exits. Calls can also be invoked asynchronously to avoid waiting on long running processes using the “async” decorator. The “stderrtostdout” and “returnstdout” decorators are used to pipe any command line standard output or errors from the remote invocations back to the STAFResult object. This is only useful for commands or applications that have command line output.

STAF will be used in the implementation of both the File Transfer and Remote Execution design components for this test automation system. It does not require storing login credentials for the various SUTs and already provides great support for all the operating systems this project required.

WEB MVC FRAMEWORK

The web2py [15] Python framework was selected for implementing the Test Automation Server. A preference to use either Python or Java for this implementation existed in the beginning due to familiarity with these two programming environments. Python was preferred due to its dynamic and interpreted [16] nature as it was expected to also be deployed to the SUTs as part of the common utilities component of the system. A compiled language wouldn't fare well in an environment like this as it would require several additional steps for implementing and deploying changes throughout the infrastructure.

Web2py is licensed under the GNU Lesser General Public License V3. It received InfoWorld's 2012 Technology of the Year award [17]. It was originally developed as a teaching tool for programming and has grown to be a full-fledged database driven web framework.

Web2py was selected as the framework for the system due to the following features that are unique to it:

- Batteries Included. The install, which is simply a compressed archive, contains everything you would need to run the framework including a Python interpreter, web server, database abstraction layer and SQLite database [18].
- Very small footprint of 1.4MB and no configuration required to run.

- Applications can be modified, installed and uninstalled without the need for restarting the web server.
- Integrated development environment that is accessible through a web browser. Editing source code and deploying applications can all be done through a browser.

The ability to easily modify and maintain the application, even from a browser, and the small and easy deployment of the framework meet the goals and requirements for this project. It promoted rapid application development and prototyping of features. This is why web2py was selected for this project over other very successful Python frameworks such as Django [19] and Pylons [20].

Chapter 5: Implementation Results, Considerations and Caveats

IMPLEMENTATION RESULTS

The implementation of the test automation system was completed in sixteen months and consumed two Engineers working on the project. The effort produced over eighty thousand lines of code and the project now contains over one hundred and ten thousand lines of code from contributions by Quality Engineers and Developers as well. Operating system coverage testing was transformed from a manual process involving a handful of Windows and Linux SUTs to over thirty operating systems exercised automatically with each iteration of a test pattern. Moreover, testing permutations with different operating system and product combinations was also expanded due to the automation of the install and removal of the software products.

This increased test coverage accelerated the defect detection rate in the organization. A recent example of this was the introduction of networking changes to the products that were compatible with all the supported operating systems except for the oldest supported Windows operating system in this group which is Windows XP. The defect was immediately detected using the automation system. A manual process would have very likely missed this defect as older operating systems might be deprioritized in manual test efforts. Having this automation framework in place increased the confidence of the organization in detecting defects which has led to more aggressive feature roadmaps for our products.

IMPLEMENTATION CONSIDERATIONS AND CAVEATS

The system implementation was divided into several phases. The first phase was the development of the product and test staging repositories with the STAF enabled file transfer and remote execution components for distributing and installing the software

product on the SUTs. Some challenges were faced while implementing two key features in this phase: operating system detection and file transfer performance.

An early implementation of the test automation system did not dynamically detect the remote system and relied on manual population of the SUT database table on the TAS. This approach was very error prone as it is easy to forget to update the table when a configuration change is performed on the SUT. Many test cycles were wasted on misconfigured tests due to this approach as tests failed to execute properly on various systems. This approach was abandoned and replaced by a dynamic method of detecting remote operating systems.

The STAF Variable service was used to dynamically detect the operating system of the SUT. The STAF system variable “STAF/Config/OS/Name” was used to determine the operating system, but it did suffer from a few limitations. Some Windows systems would return “Unknown WinNT” as the result of the variable query instead of a specific operating system. This for the most part didn’t affect the automation effort as the family of the operating system, in this case Windows, could still be deduced which, was more important than the specific version. However, on Linux the variable always returns a generic “Linux” result. A few more queries would be required to determine the specific operating system installed if that information is important to the automation effort. On Linux, using the File System Service to retrieve the “/proc/version” file and parsing can typically provide specific distribution information, however that is not guaranteed. The “/etc” directory also typically has specific distribution files that can be queried such as “suse-version”, “redhat-version”, “fedora-version” and so on. On Windows, access to the system registry is required to determine the specific operating system edition. The registry key “HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\ProductName” will contain the specific edition of the Windows

operating system. The STAF Process Service would be required to execute the command line “reg.exe” utility to retrieve the value of the key. A similar approach for Macintosh OS X would be required; the STAF Process Service would need to be used to execute the “sw_vers” utility to retrieve the product name and version.

As for the file transfers using the STAF File System Service, the speed of the file copies were roughly 70% the speed of a traditional file transfer using SAMBA or Windows Shares. This might not be an issue if the products, tests and test results aren't large data sets. File transfer durations were a small percentage of the total time spent configuring and executing test cases in our environment. Therefore the performance of the system wasn't largely degraded by using STAF File transfers and was deemed acceptable when weighed against the advantage of having a uniform file transfer solution.

The second implementation phase was for the Standalone Regression testing execution and results gathering. Implementation items to consider here are the use of conventions for common test attributes, such as locations of test binaries in the staging repository and their respective locations on the SUTs once they are copied over. This helps in reducing the implementation logic significantly. Conventions for the results collection are also important to simplify the process of obtaining and storing the results along with locating the results for analysis in the future. Exercising good logging practices here is crucial for the success of the system. Each test procedure can go through many steps before actually conducting a test. Logging the results of each step leading up to the execution of the test, then the cleanup steps that occur after are all crucial for troubleshooting issues that could occur in the system. Missteps in early test harnesses can have ramifications on the execution of the other test harnesses down the line, and troubleshooting these issues without proper logging is very difficult and time consuming.

The remaining phases of implementation were devoted to the remaining test patterns such as the Client-Server Regression test pattern. The main implementation consideration for the multi-system test patterns is how to perform synchronization between the SUTs. Serializing the execution of the steps among the various SUTs is a valid approach for implementing synchronization since it is very simple to implement and is especially viable in environments that have short running steps that lead up to the test execution. If the pre-test execution steps and post-test execution steps only take a few seconds to complete, then investing in a parallel execution system would not save on total test time and would increase the complexity of the system. However, investing in a parallel execution system by utilizing the STAF Semaphore Service is necessary if the pre and post-test execution steps take a long time to complete. Total test execution time would grow linearly with every additional SUT added to the test.

STAF AUTOMATION LESSONS LEARNED

Two issues using STAF, consistency in detecting remote operating systems and file transfer performance concerns, have already been discussed in the previous section “Implementation Considerations and Caveats”. These however were only a part of the issues discovered in our utilization of STAF. There were also other important STAF automation caveats that we experienced while implementing the system.

Problems were encountered with the STAF Process Service for invoking remote commands on x86_64 Windows systems that were running an x86 version of STAF. The Windows File System Redirector and Registry Redirector [21] redirects access calls to 64 bit file system and registry resources by 32 bit applications into an equivalent 32 bit resource instead. Therefore direct access to 64 bit system utilities and the registry hive is not possible from 32 bit applications, which are common steps in test configuration. This

behavior can be corrected by disabling this Windows feature. However, disabling this feature can have an impact on the software product being tested. A better workaround is to always ensure that x86_64 Windows operating systems are always configured with an x86_64 version of STAF to avoid redirections by the Windows Redirector.

Problems were also encountered with starting the STAF client on Linux systems that have statically assigned network addresses. The STAF client fails to retrieve a hostname in this configuration which leads to a failure in initializing the client. The STAF client works properly on Linux systems that are configured to use DHCP and DNS servers. A workaround for the problem on systems with static network addresses is to add an entry for the local host in the “/etc/hosts” file.

And finally, the STAF client provides a command line utility for invoking STAF calls. This utility is very helpful for quick proof of concepts and troubleshooting. I however would recommend avoiding implementing the automation framework using direct invocations of this tool. It might seem faster at first to do so, but implementing your own parser for the command line output and maintaining it are steps that the STAF framework already provides you programmatically. This could also introduce defects in STAF result interpretation by mishandling unexpected return codes from the STAF calls.

Chapter 6: Conclusions

Implementing a software test automation solution for a heterogeneous test environment is a very large undertaking. A large amount of resources and time can be consumed if careful planning isn't conducted to avoid common automation pitfalls. The rewards from undertaking an effort like this might not materialize immediately, but the increased coverage of product testing will undoubtedly increase the quality of the software product.

Establishing a system that integrates seamlessly with existing business processes is essential for the adoption of an automation solution. Leveraging existing and well established automation tools and development frameworks will accelerate the implementation of the system and increase the chances of success. These tools and frameworks can be open source and freely available which also minimizes the overall cost of the automation effort.

The tools selected for this automation implementation, STAF and web2py, do come with caveats and challenges. I however still prefer implementing automation solutions using these tools over building an in-house solution or purchasing a proprietary solution. STAF and web2py provide tremendous features straight out of the box and they both have vibrant support communities.

The high level design described in this report can be leveraged by any organization that is willing to implement an automation solution for a heterogeneous environment. The core design concepts, data models and implementation considerations will hopefully assist these organizations in their automation endeavors especially if they use STAF in their automation implementation.

FUTURE WORK

There are a couple of items to investigate for future improvements of this test automation infrastructure. The first concerns the file transfer performance issue mentioned previously. A mitigation plan is needed if this becomes a problem in the future. STAF provides additional add-on services that are not included in the default installation of the utility. One of these additional services is an FTP Service [22]. File transfer performance tests should be conducted using this service to see if any improvements are detected. Another option is to build an in-house solution for transferring files in the test environment.

Another area of future work is to investigate if this design and implementation can be leveraged to automate testing in cloud environments such as Amazon Web Services [23]. The demand to conduct testing on cloud environments, especially Client-Server Regression testing, will increase as more businesses adopt cloud technologies.

Bibliography

- [1] Persson, C.; Yilmazturk, N.; , "Establishment of automated regression testing at ABB: industrial experience report on 'avoiding the pitfalls'," Automated Software Engineering, 2004. Proceedings. 19th International Conference on , vol., no., pp. 112- 121, 20-24 Sept. 2004
- [2] Ramler, R.; Klaus Wolfmaier, K.; , "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost." In Proceedings of the 2006 international workshop on Automation of software test (AST '06). ACM, New York, NY, USA, pp. 85-91
- [3] Microsoft Windows Product Support Lifecycle, <http://windows.microsoft.com/en-US/windows/products/lifecycle>
- [4] Red Hat Linux Support Lifecycle, <https://access.redhat.com/support/policy/updates/errata/>
- [5] Suse Linux Support Lifecycle, <http://support.novell.com/inc/lifecycle/linux.html>
- [6] Test Harness, http://en.wikipedia.org/wiki/Test_harness
- [7] Buschmann, F.; Henney, K; Schmidt, D; , "Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing." John Wiley & Sons 2007.
- [8] Model-View-Controller, <http://en.wikipedia.org/wiki/Model-view-controller>
- [9] SSH IETF Request For Comment 4253 Website. <http://tools.ietf.org/html/rfc4253>
- [10] RSYNC Utility Website. <http://rsync.samba.org/>
- [11] SAMBA Website. <http://www.samba.org/>
- [12] Rankin, C.; , "The Software Testing Automation Framework," IBM Systems Journal vol.41, no.1, pp.126-139, 2002
- [13] STAF Main Website, <http://staf.sourceforge.net/>

- [14] Cervantes, A.; , "Exploring the use of a test automation framework," Aerospace conference, 2009 IEEE , vol., no., pp.1-9, 7-14 March 2009
- [15] Web2py Web Framework Main Website, <http://www.web2py.com/>
- [16] Python Overview Website, <http://www.python.org/about/>
- [17] InfoWorld's 2012 Technologies of the Year Award Winners,
<http://www.infoworld.com/slideshow/24605/infoworlds-2012-technology-of-the-year-award-winners-183313#slide23>
- [18] SQLite Database Website, <http://www.sqlite.org/>
- [19] Django Python Web Framework Website, <https://www.djangoproject.com/>
- [20] Pylons Python Web Framework Website, <http://www.pylonsproject.org/>
- [21] Windows File System and Registry Redirectors,
<http://msdn.microsoft.com/en-us/library/windows/desktop/aa384249>
- [22] STAF FTP Service, <http://staf.sourceforge.net/current/FTP.html>
- [23] Amazon Web Services, <http://aws.amazon.com/>