Avalanche: On-Demand Virtual Environments for Automatic System Testing

by

Helge Mahrt

A thesis submitted in partial fulfilment of the requirements for the degree

of

Diplom-Informatiker (Berufsakademie)

in the Graduate Academic Unit of

Applied Computer Science

at the

Berufsakademie Stuttgart

September 2008

Duration	4 months
Course	TIT05AIA
Company	Hewlett-Packard Ltd. Bristol, United Kingdom

Examiner at company	Peter Toft
Examiner at academy	Prof. Dr. Karl Friedrich Gebhardt

Avalanche: On-Demand Virtual Environments for Automatic System Testing

Helge Mahrt

August 31, 2008

Abstract

Distributed applications require distributed testing. There are many challenges regarding synchronization, orchestration and verification of tests in a distributed environment. In this thesis a system is introduced which is designed to run and verify tests automatically across a distributed environment, using virtual machines as the distributed nodes. The system makes it possible to integrate distributed system testing into an automated testing cycle.

Contents

List of Figures

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
RMI	Remote Method Invocation
SmartFrog	Smart Framework for Object Groups
SUT	System Under Test
VAST	Virtual Environments for Automatic System Testing, the system described in this thesis.
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

Due to hardware limitations, the current trend in computer architecture isn't to scale up processing power anymore. Faster processors would cost too much to be paying off. Instead the latest development is to use more processing cores - whether on the same motherboard or within a distributed system.

Therefore distributed computing and concurrency are now large topic. Applications intended to run in large scale systems have to be designed specifically to work efficient on multiple machines. Hence the tests for those applications have to be written accordingly - to work in a distributed environment, too.

Testing has its own trends as well and the latest one is called *continuous integration*. To be more precise, it is not only about tests, but the whole software development cycle. In *continuous integration* tests are not executed manually, at one point, to see if an application is working, but automatically. There are now build systems (Hudson [Hud] and CruiseControl [Cru] are two examples), running in a clone of the production environment or at least in

a clean virtual machine, which periodically check out the latest version of the application from a code repository, build it and run its tests.

Developers will be notified as soon as tests start to fail - which means the build systems do automatic regression testing as well.

In this thesis I will describe:

- A system for distributed testing based on virtual machines (section 3.3) and its setup. (section 3.4)
- The design and the structure of testcases. (section 3.5.2)
- The orchestration of the tests (section 3.5.3) and the verification of their results. (section 3.5.4)

Finally I will draw a conclusion and describe further work. (section 5)

Chapter 2

Application

A key challenge in testing distributed applications is to orchestrate tests across the distributed system. It has to be ensured that the appropriate tests are executed on the appropriate machines. Additionally, the single nodes must not be regarded uniquely but as a whole system. Therefore the state of the complete system has to be observed to verify test results. But this leads to the "State Explosion Problem" [Val98]: It is impossible to regard all attributes in a distributed system as the number of states grows exponentially with an increasing number of nodes.

I chose a challenging example as a distributed application to act as my System Under Test (SUT) . It consists of three nodes. Two of them are capable of running applications and possess a database each. Both databases are synchronized and contain the applications' current state at any time. The third node acts as an observer.

Applications will be running on one node only. There is a fourth node in the system which merely redirects outside systems to the machine running the applications.

In case of a failure of one node the other two will detect this and take according actions. If the node running the applications fails, the other node with a database will take over. It will start the applications again and restore them to the state stored in the database. The access point for outside systems will be notified and update its redirection.

The purpose of the observer is to detect if there was just a network failure between the other two nodes, or if one of them really went down. Without a third observation point they wouldn't be able to tell and therefore couldn't decide which role to assume - whether to run the applications or not.

This SUT is built to be fault tolerant and to guarantee a limited amount of application downtime (the time it takes for another node to take over), so a key part of testing the application is to test its behaviour in the presence of failure. For this purpose a distributed environment, to run the SUT on, and a testing application are required. Failures have to be simulated and the behaviour of the SUT has to be verified:

- Do the remaining nodes take the correct actions when one node fails?
- Are the applications started correctly on the intervening machine?
- Is the service/application still reachable from outside after a failure?

These are only a few questions which have to be converted into (more specific) testcases. Besides being able to run on a distributed system, the testing application is required to have enough control to simulate errors like a network failure or the sudden killing of one of the SUT's processes.

Chapter 3

Virtual Environments for Automatic System Testing

3.1 Concept

To take advantage of the possibilities the current virtualization technology offers the test environment should be built upon virtual machines. This has the advantage that - theoretically - a distributed system of any desired size can be tested. Practically there will be a limit to the number of virtual machines per physical machine, because, once the capacity of a physical machine is fully utilized, adding more virtual machines will only decrease the average performance of them.

Using virtualization also allows to create the system from scratch for each test run. This ensures that the machines are in a clean state without being influenced by previous test runs.

After finishing the construction of the virtual environment the necessary software components will be distributed across the system. Then the tests will be orchestrated according to a test suite, which is modelled after a reduced state machine. Sequences of actions will be executed and verified by the outcoming states. Actions are not only invoking SUT functions but also inducing errors to the system, to test the fault tolerance.

Virtual Environment for Automatic System Testing (VAST), is designed to run completely automatically, which enables it to be part of a *continuous integration* cycle. It could be integrated in the repetetive test cycle of systems like CruiseControl or Hudson and add the feature of automatic system testing to them.

3.2 Basics

3.2.1 SmartFrog

> "SmartFrog is a powerful and flexible Java-based software framework for configuring, deploying and managing distributed software systems." [Sma]

SmartFrog (Smart Framework for Object Groups) essentially consists of two parts: The SmartFrog language and the SmartFrog daemon.

The language is used to describe software components, to define which attributes they possess - basic datatypes, like integers, strings etc. or references to other components. The components are linked to Java classes which have to implement the necessary interfaces to enable the life-cycle control and the remote method invocation[1] (RMI). RMI is used for method calls between components, which means it doesn't matter where components are running - they are accessible in the same way when running on the same machine as on different ones.

The SmartFrog daemon is the runtime system. It interprets description files, initializes and starts components accordingly. It manages their life-cycles and enables components to communicate with each other. It is even possible to deploy components to daemons running on remote machines. [GGL+03]

[1] http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

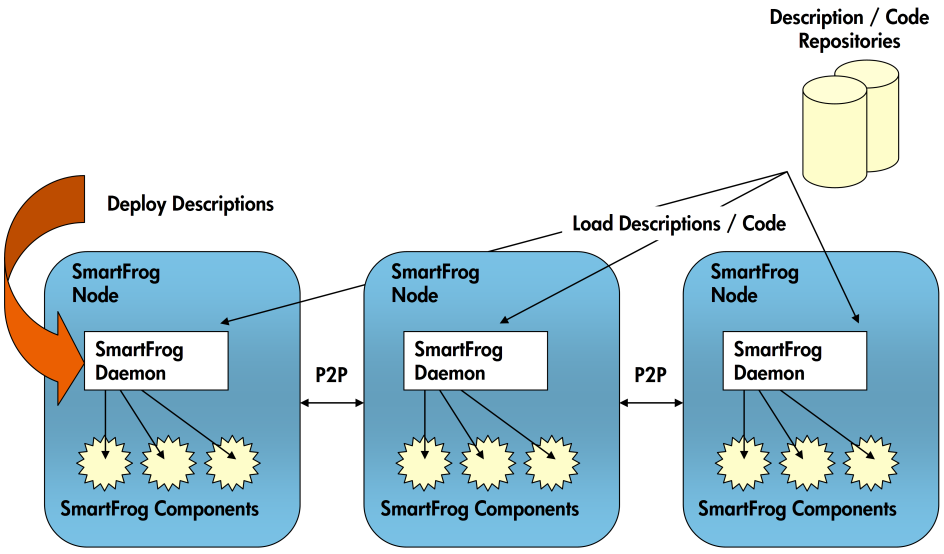

Figure 3.1: SmartFrog.

3.2.2 Avalanche

Avalanche is a distributed deployment and change management system based on SmartFrog. It controls the complete life-cycle of software - respectively components - on remote machines, from the deployment and installation up to the eventual termination. [SG08]

The most apparent difference to SmartFrog is that Avalanche doesn't require a SmartFrog daemon running on the remote machines to deploy components onto them. It only requires access via SSH+SCP or Telnet+FTP. During the so-called host "ignition" Avalanche transfers an installation package to the remote hosts and sets up the SmartFrog daemon. When the daemon is running the chosen software components will be initialized and started. It reports back to Avalanche via XMPP.

Avalanche requires an external XMPP server to communicate with the

remote hosts.

Originally Avalanche was only controllable from its website but for the purpose of using its functionality programmatically it is now available as a SmartFrog component, too.

Recently Avalanche has been enhanced by a component, which is able to handle virtual machines using VMware Server, and therefore is now able to create and control virtual machines on remote machines. [Mah07]

3.3 The Virtual Environment

3.3.1 Virtual Machines

Virtualization Technology

Because the virtualization component, used by Avalanche, uses VMware Server, this is the virtualization technology used here as well. The author of the component chose VMware Server over other technologies like Xen[2] because it is freely available, takes less effort to install than others and is platform independent. [Mah07]

Template Machines

Creating virtual machines from scratch would be too much effort. It would require to automatically install an operating system, configure it in the right way and install the minimum required software - a difficult process with lots of possibilities to fail. Therefore prepared virtual machines are being used as templates. They have to fulfill the basic requirements described in the next section.

When chosing to create a virtual machine one chooses from the template machines and Avalanche will create a copy - including a copy of the virtual hard disk. To prevent this operation from happening more often than necessary snapshots are being used.

Snapshots are images of the current state of a virtual machine. (The state of the hard disk and the memory.) Using them it is possible to restore a virtual machine to this exact state again.

The first time a virtual machine is created, before starting it, Avalanche

[2] http://www.xen.org

will take a snapshot of the "clean" state. Everytime the system tries to create a virtual machine it checks if there is already existing one with the same name. It will rather load the snapshot of an existing machine than copy the whole virtual hard disk and the configuration files again, which speeds up the environment construction when doing repetetive tests.

Requirements on the Virtual Machines

The template virtual machines, which are being used to create the nodes of the System Under Test, need to be prepared before using them. Any required software, additional to that for Avalanche (e.g. databases, services etc.), should be installed and configured so that it is ready to use.

Besides that the virtual machines need two network interface cards. As described in section 3.3.4 this is essential for VAST to work, but easily accomplished thanks to virtualization.

Because it would be difficult to have customized network configurations on the template machine the network setup helper (see section 3.3.4) will do the job of configuring the network cards when the environment is constructed.

There isn't (yet) a central repository for the virtual machines so the (prepared) template machines have to be copied manually to every physical host which should be able to use them.

3.3.2 Test Runner

The test runner will be running on each node of the SUT. As mentioned, in a separated SmartFrog daemon. It is responsible for executing actions according to the commands it receives via the broadcast communication. (See section 3.3.4.) E.g. starting scripts, invoking methods with a given set of parameters or publishing modified attributes. It is also responsible for cutting and restoring the network connection of the SUT.

3.3.3 Test Controller

The controller is running on the single virtual machine which is not part of the SUT. It has complete knowledge about the state machine (see section 3.5.1) and uses this data to orchestrate the tests (see section 3.5.3). It sends commands to the runners on the dedicated VAST network and checks the resulting states by querying their SmartFrog daemons for the attribute values. The test controller constantly reports the progress of the system tests to the Avalanche server using XMPP.

3.3.4 Network Communication

One task of the system testing environment is to simulate - to inject - errors and observe the behaviour of the SUT. A very common fault in a distributed system is an interruption of the network communication and a common counter measure is a fail-over behaviour, where a different machine in the system adopts the role and the state of the failed or separated one. Simulating a complete cut of the network connection to the SUT entails the problem of losing connectivity to the test runner and therefore losing control of the node.

Just trying to isolate the software and preventing it from using the network interface card is difficult as you still have to maintain the connection for the test runner. (E.g. it is not possible to just remove the default gateway because that will cut the connection for all applications.)

But using virtual machines offers a solution: virtual network interface cards. Template machines for the VAST environment are required to have at least two network interface cards, one for the SUT and one for the test runner. Having VAST communicate on a separated network using separated network adapters enables it to completely cut the network connection of the SUT without disabling itself. At the same time it minimizes the risk of the isolated SUT node still being able to communicate with the rest of the system.

This requires the Avalanche server, or at least the test controller, to be able to communicate on the VAST network. Because it is not unlikely that there is only one network card present on the machine of the Avalanche server the easiest and most convenient solution is to have the test controller run in a virtual machine, too. It will communicate with the test runners on

one network card and with the Avalanche server on the other. An illustration of the networks can be seen in figure 3.2.

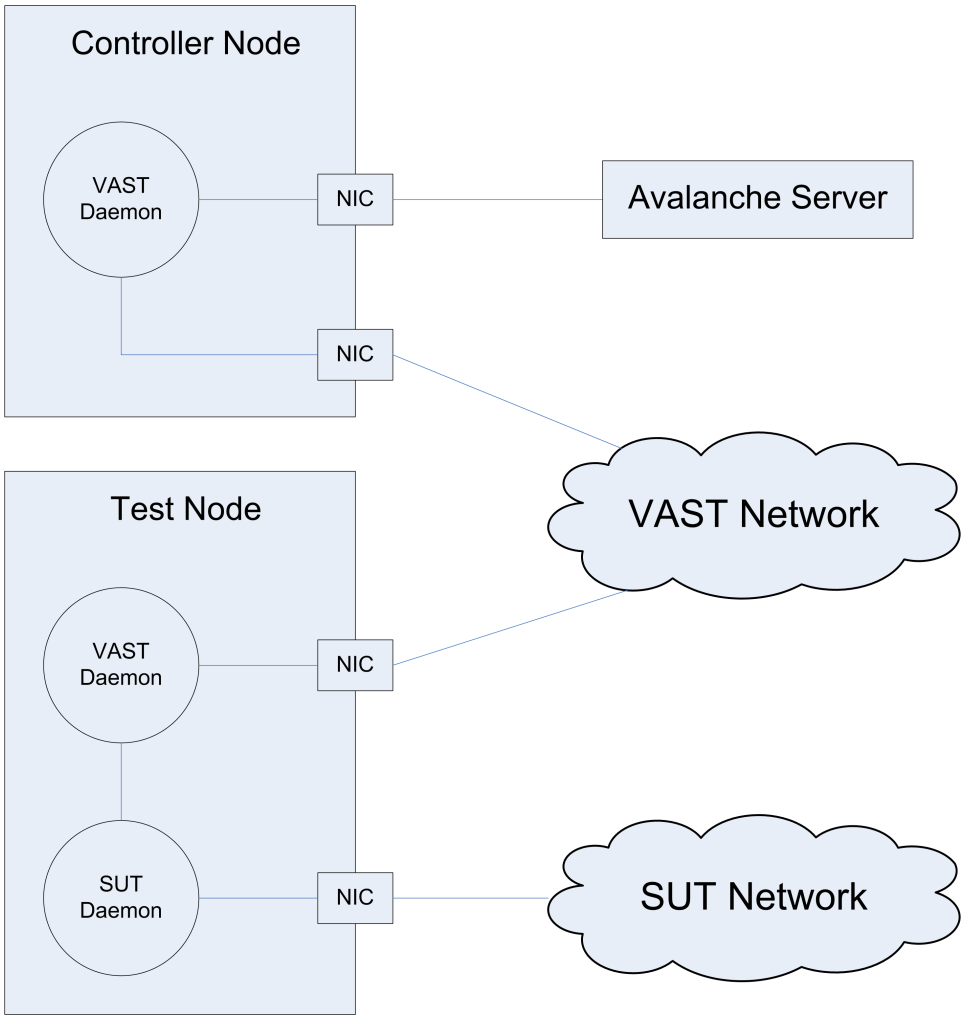

Figure 3.2: An illustration of the VAST networks.

Network Setup Helper

After powering on the virtual machines they need network connectivity so Avalanche can take control of them. Because it is possible that there is no DHCP server present in the network, the configuration has to be done manually. Using the VIX API, which is used by the virtualization component to interact with VMware Server [Mah07], it is not possible to configure network cards within a guest operating system[3].

But it is possible to execute programs and to copy files into the virtual machine. This is where the Network Setup Helper comes into play: when the guest OS has come up and the VMware tools[4] are running, the helper will be copied and executed. Given the necessary parameters it will set up the network connections. Additionally it will set the default gateway and insert static routing addresses for the virtual machines which are part of the testing environment. (It is also possible that there is no DNS server present in the network.)

VAST Protocol

VAST uses its own protocol, the messages being sent as UDP broadcasts. The reason for that will be explained in more detail in section 3.5.5. The protocol is based heavily on the Strategy Pattern[FFSB04]. Each message calls its respective method in an interface, which will be implemented by the test controller and runner. (See figure 3.3 for an illustration.)

This makes the protocol flexible - because classes implementing the interface can implement different behaviours - and easy to extend. Adding a

[3]Guest OS - The operating system running within a virtual machine.

[4]VMware tools - The tools need to be installed on the guest OS for certain operations. (e.g. copying files, executing applications etc.)

new command to the protocol requires just a new class, which implements the message interface, and an additional method in the callback interface - previously existing code is not affected.

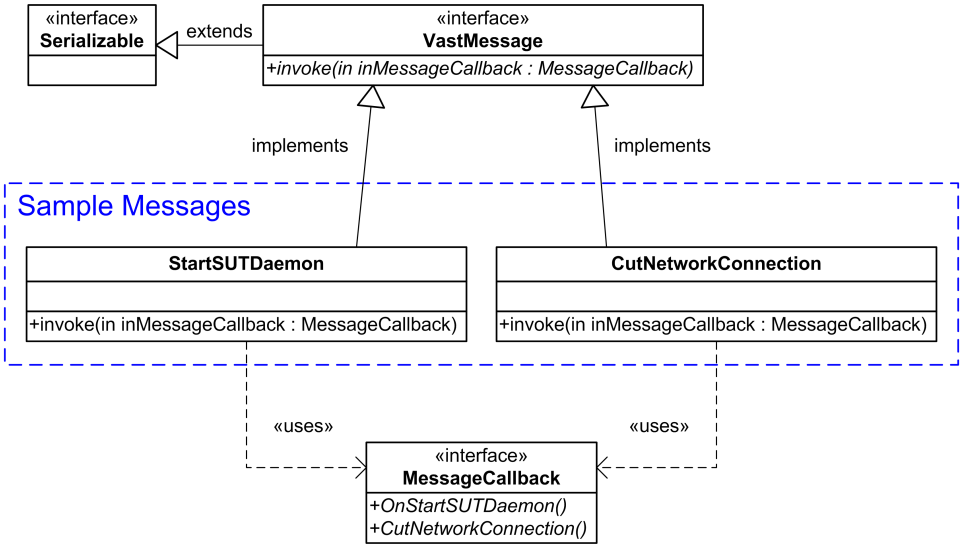

Figure 3.3: The class design of the VAST network protocol.

Running two Daemons

The SUT should be as little influenced by VAST as possible. Therefore two SmartFrog daemons will be running on the nodes of the SUT. One for its components and one for the VAST test runner. This way it is ensured that the components of the SUT are running on a clean daemon. Another reason for having two daemons is the networking: The daemon for the SUT should be listening on the network connection for it and the daemon for the runner should be listening on the separated VAST network.

During the ignition of the nodes the daemon that is started by Avalanche will be used for the test runner. The test runner then takes the installation package of Avalanche and extracts it a second time to a different place - without the VAST libraries of course - and adds the libraries of the SUT package. The second deamon will be started directly but the test runner waits for commands from the test controller before it starts any component belonging to the System Under Test.

3.4 Environment Construction

3.4.1 Machine Descriptions

Each machine, physical or virtual, is described in the SmartFrog language. Before Avalanche was available as a SmartFrog component it was necessary to enter a certain set of configuration parameters on the Avalanche website. Because VAST accesses Avalanche programmatically - for automation purposes - those values need to be defined in the description files.

Each physical machine description consists of the basic parameters, the host address, the processor architecture (x86, IA64), the platform (Intel, AMD etc.) and the operating system, as well as the environment variables and the username-password pairs for transfering the installation packages. Due to the abilities of the SmartFrog language it is very easy to extract and reuse common parameters, which most or all machines share.

Virtual machine descriptions are derived from the physical machine description, because they need the same parameters for Avalanche, plus additional parameters required for setting up the virtual machines. Some notable ones are e.g. the affinity, which defines on which physical machine the described virtual one should be created and run, or the source image, which defines which virtual machine template should be used. It is also necessary to define the network parameters for the dedicated VAST network and to set the user credentials to access the guest operating system running on the virtual machine.

3.4.2 Preparing the SUT

The System Under Test has to be prepared for the testing so the test runner is able to invoke functions and publish attributes of the components. Therefore the relevant classes have to be subclassed. Each function which changes an attribute has to be overwritten, to change the value of the published attribute in the SmartFrog daemon as well. Functions which should be invoked by VAST do not necessarily need to be overwritten, except there is a need for parameter conversion or preparation code.

This entails the requirement, that the System Under Test is written in Java. If it wasn't designed to be running as a component within the SmartFrog daemon, the necessary interfaces have to be implemented. The initialization and the start of the application have to be called according to the lifecycle of SmartFrog. Additionally a description file written in the SmartFrog language is required for the daemon.

The SmartFrog scripts, which start the SUT, have to be modfied or replaced by ones that use the subclasses instead of the original ones. This is easily done as the SmartFrog language allows prototyping: you just have to extend the original description and replace the reference to the base class with the reference to the derived class.

Additional to the preparation of the virtual machines (described in section 3.3.1), packages for the SUT, which contain all required jar files, need to be prepared. They should contain all code required for and by the SUT. In section 3.4.3 it will be described how these packages are used for the System Under Test.

3.4.3 Construction Sequence

The construction sequence consists of two parts: The first part is the sequential execution of the necessary steps to prepare the ignition of the physical hosts. It starts with parsing the information about the physical and virtual machines, which will then be stored in the Avalanche database. Next the packages, which will be used to ignite the virtual machines, are being generated. They contain the SmartFrog daemon and all necessary files for the System Under Test and for the system testing.

After the ignition of the physical machines has been set off the construction sequence becomes a event driven process: The listener is waiting for messages from the hosts and the environment constructor will react accordingly. First it will wait for all physical machines to start, before trying to restore the virtual machines to their snapshots - if there are any, machines and snapshots. If it fails to do that it will create a new copy of the relevant template virtual machine, take a snapshot and then start the copy. If reverting to the snapshot succeeds it will directly start the virtual machine.

Subsequently the environment constructor will send a message to the virtualization component on the physical machines and tell it to wait for the VMware tools to start in the guest OS. As soon as the tools have started the component notifies the environment constructor. The network setup helper will be copied into the guest OS and executed. After setting up the network connections the virtual machines will be ignited by Avalanche, too.

When all virtual machines have been ignited and they log in to the XMPP server and reported their presence the environment constructor will send the start signal to the test controller and the system testing begins.

3.5 Running and Verifying the Tests

3.5.1 Reduced State Machine

A state machine is an abstract view of a system consisting of states and actions that lead the system to translate between the states.

The "State Explosion Problem"[Val98] is a combinatory problem which describes the rapid growth of states in a distributed system as it scales up. The number of states grows exponentially with increasing number of machines. Therefore VAST uses a reduced state machine.

Reduced, in Reduced State Machine, means that only the relevant attributes are used to model the state machine and to verify a state - all other values are ignored. This reduces the number of states and makes them more manageable but at the same time means that you have to know which attributed are important for a state when you create the tests. Forgetting about an attribute might lead to false state verification and thus to false test results.

3.5.2 State Machine Description

The state machine is described in the SmartFrog language and running as an independent component. This is because of the simplicity to resolve attributes: Instead of letting the component that wants to use the state machine parse the complicated nested descriptions it is much easier to have every attribute, state, action etc. resolve their own set of values and gather them in the test suite.

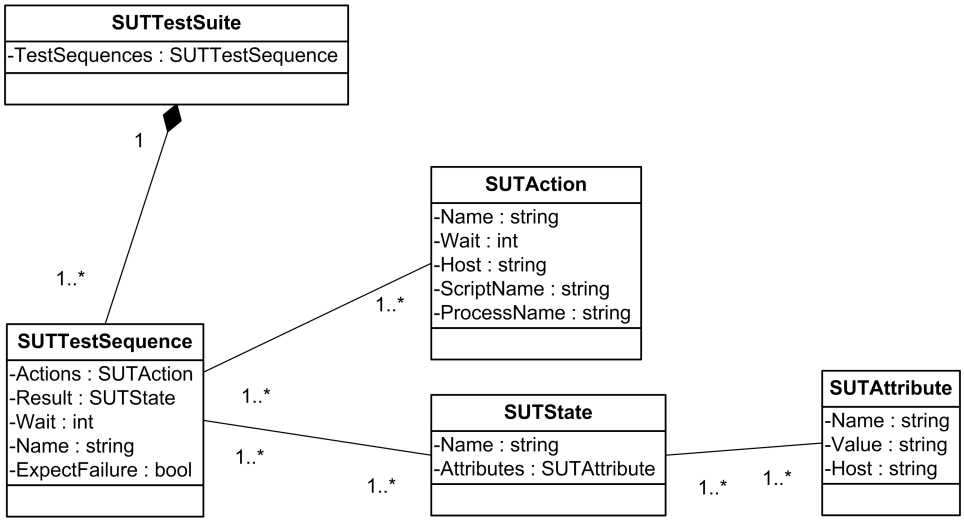

Figure 3.4: The classes of the State Machine in VAST.

Note that all relations between the classes in figure 3.4, except one, are many-to-many relations. This is because of another advantage of realizing them as self-resolving components: Using the SmartFrog language you can decide whether a reference to another component is really a reference to that unique instance or if it is a reference to a *duplicate*. In this case real references are used, to save memory and because attributes, states etc. are reusable, which means there is no need for copies.

Attributes are the exposed values of nodes in the distributed system which are used to verify states. Each attribute consists of a name and a value. The description states the expected value and on which machine the attribute can be found. The SmartFrog daemon on that machine contains the attribute with its actual value.

States define a partial/reduced view of the distributed system. They consist of a list of attributes and a name to identify them in the log messages.

Actions are carried out to translate from one state to another. There are two types of actions: Executing a function of the SUT and inducing failures. The latter are executed by special functions which can be called via the VAST network communication, too. (E.g. cutting the network connection of the SUT.)

Sequences are series of actions with one resulting state. The state is used to verify that the System Under Test behaved appropriately when the actions were executed. Each sequence also possesses a wait attribute which defines how long the test controller should wait, after executing the actions, before it tries to verify the resulting state.

There is an additional variable to each sequence which defines whether it is expected that the sequence produces a failure or not. If a failure is expected and there is one the sequence will be passed as successful.

Note that the consecutive sequence starts where the previous one finished. If there was a failure (expected or not) the following sequence should be modelled accordingly.

Test Suite The test suite is the main entry point for other components. It contains the list of test sequences which can be used for the orchestration of tests as described in the next section.

3.5.3 Orchestrating the Tests

As soon as the test controller receives the signal from the Avalanche server to start the testing it will read the first test sequence of the state machine and execute the actions sequentially. For each action it sends a command to the appropriate machine. If there is a wait time given it will wait accordingly before executing the next action. After all actions of a sequence have been performed the controller will wait for a defined time for the system to reach the desired state before trying to verify it.

The controller will send a message to the Avalanche server reporting the result of the test sequence. This report contains the name of the sequence as well as the attributes and their values, which have been checked. The server will be notified of the actions which are induced as soon as it happens, so there is no need to report them again.

After all test sequences have been passed the system will not be shut down, because it might be required to examine the nodes of the SUT manually (e.g. for local log files).

3.5.4 Verifying States

As described earlier, a state is a simplified view of the system consisting of several attributes. To verify a state those attributes have to have the appropriate values.

I decided to use a polling mechanism where the test controller queries the appropriate nodes of the SUT for the required attribute values when verifying a state.

Additionally, whenever there is a change made to an attribute on a node, that node will send a broadcast on the VAST network. This has the purpose of increasing the logical clock (explained in section 3.5.5) and to have the Avalancher server log that information. When reconstructing the order of events it is then possible to see when which changes to attributes have appeared in the system.

3.5.5 Logging of the Events

When testing software it is essential to know what happened during runtime. Therefore log files are a good solution to keep track of the events and the order in which they happened. This is not as simple in distributed systems as it is for single threaded applications and poses some challenges. One of them is the ordering of the events - a solution for that is described in section 3.5.5 - and another one is gathering the output of all machines at one single point and displaying it in a suitable way so users can extract useful information.

Each machine belonging to the VAST system will keep a local log file which logs the output of the SmartFrog daemons and the running components. These can be used in cases of unexpected failures to determine its source. Additionally the test controller will send event messages to the Avalanche server which will store them in its database.

Because it is very useful to be able to see in real time what is going on in the SUT the Avalanche website is able to display the event messages as soon as they arrive. They will be ordered using to their logical timestamp and inserted into the displayed list accordingly using AJAX.

Ordering of Events

There are several unknown factors when trying to determine the order in which events happened in a distributed system.

First of all, there is the delay of the network communication. It is impossible to say precicely how long it will take for a message to be delivered. The best approach is to try to approximate the time and guarantee limits within messages are received. But still there are problems in a distributed

system where events and reactions to these events happen very rapidly.

Second, for an ordering according to real time it would be necessary to have exactly synchronized clocks on each and every machine in the distributed system - and to keep them synchronized, because usually physical clocks tend to drift apart as some run a little slower or faster than others.

The third factor is only relevant for (distributed systems using) virtual machines. Time does not pass linearly within them, sometimes time even stops and "catches up" after a while. This effect is caused by the scheduling of the underlying operating system: Time passes only if the virtual machine is assigned cpu time. If it is not receiving cpu time for an exceptionally long period it appears as if time stopped within the machine and causes it to be delayed.

Luckily VAST does not require a real-time ordering of the events, a total ordering is adequate. Therefore a logical clock is being used.

Logical Clocks

A logical clock is based on virtual time. It is an event driven time which does not advance on its own but everytime an event happens. Events can be process-internal only or sending and receiving a message. The Lamport timestamp [Lam78] is the first and most basic logical clock. According to it there is one single time counter for the complete distributed system. Each node has a local approximation of this global time which is used as a timestamp for events. Using the timestamp it is possible to give an approximation of the order of events afterwards.

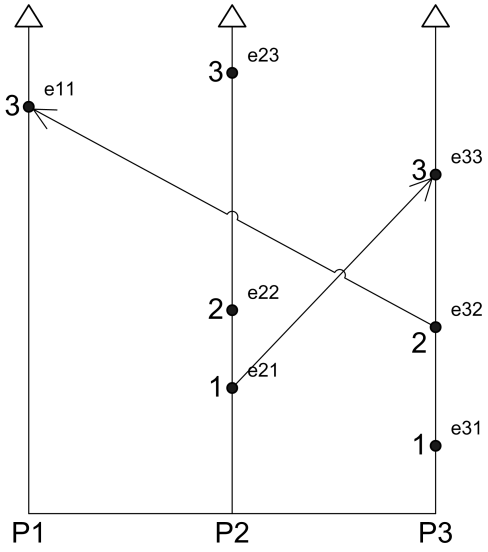

Figure 3.5: A Lamport timestamp.

Figure 3.5 shows an example of a Lamport timestamp in a distributed system with three processes. The vertical arrows show the increasing time, points denote events and the black arrows represent messages.

Vector clocks, introduced by Friedemann Mattern [Mat89], use an independent counter for each process of a distributed system. The global timestamp is therefore represented as a vector, with the counter values of all processes as elements. He describes how to make better assumptions about the causal relation of events using this method instead of the Lamport timestamp.

The Logical Clock used in VAST

As mentioned above, vector clocks are best suited when it comes to distributed debugging, where it is important to know about the causal influence of events. But to fully utilize them they have to be tightly integrated into the System Under Test so that each message sent contains the vector timestamp.

VAST is not a system for distributed debugging. It is designed to enable distributed system testing, without intruding on the SUT. VAST needs connections to the code of the system, but that is realized by using subclasses which only act as notifiers for VAST - the original behaviour remains unchanged. Therefore it is not possible to use a vector clock to its full extent - the test runners do not communicate with each other. Only the controller exchanges messages with them.

The lack of communication would mean to have a separated, independent counter for each node without any relation to the remaining system. If the runners would report their time back to the controller, and the controller would distribute it to the other nodes, it would look like there would have been a causal relation when there was none. The test controller orchestrates the tests solely based on the state machine and the reports it receives from the runners are just being logged and then discarded. That means a message sent from a runner to the controller has no influence on the behaviour of the controller.

This does not violate the assumptions made about the vector clock - because it only says that there *could* have been a causal influence, not that there *was* one - but it also makes using vector clocks, in this case, pointless. For the same reason using the Lamport timestamp would make little sense,

as the result would look like displayed in figure 3.6. The counters on the nodes would just tick indepedently without any relation to the real sequence in which the events happened. It would not even be possible to give an approximation - and that is the sole purpose of the logical clock in VAST.

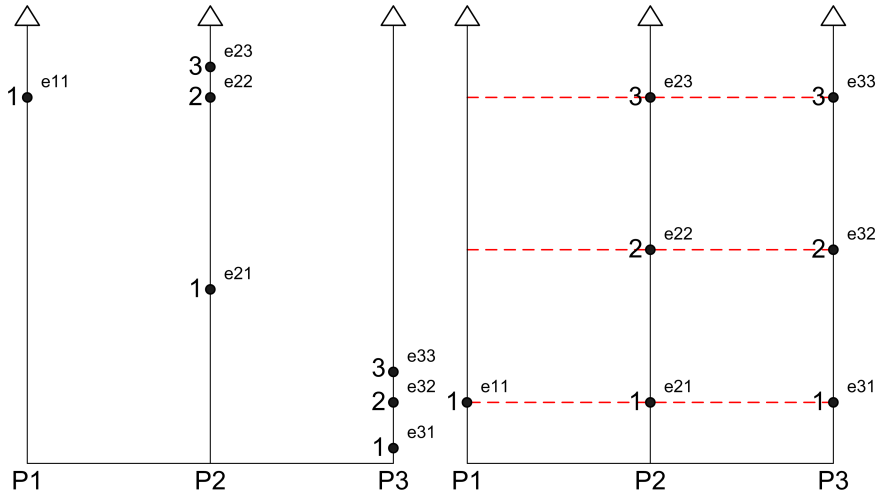

Figure 3.6: Using the Lamport timestamp in VAST would result in a reconstruction without any relation to the actual order of events.

Therefore I decided to announce the events via broadcast messages to the whole distributed system. The other nodes can then adjust their timestamp as shown in figure 3.7. The reconstructed sequence then would look like figure 3.8.

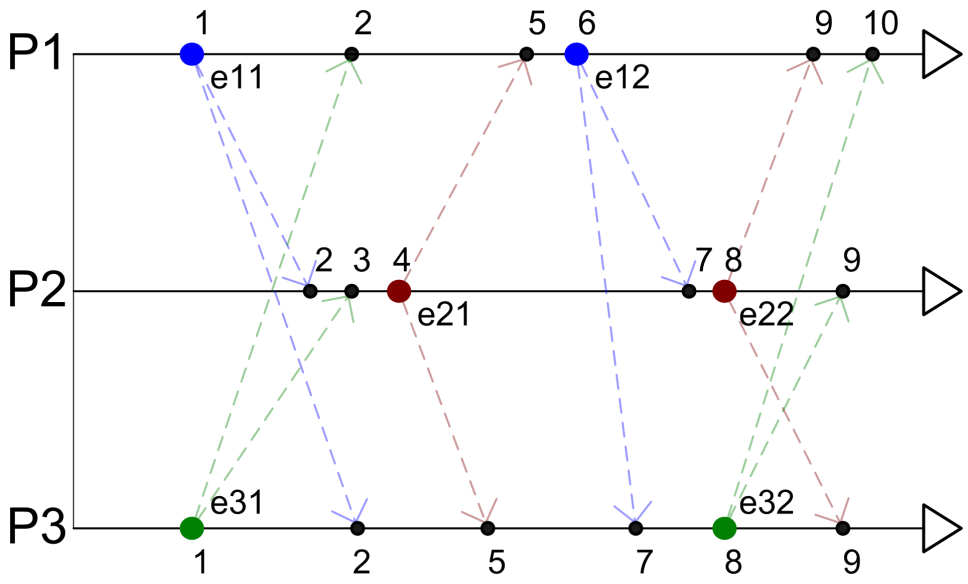

Figure 3.7: Announcement of events using broadcasts.

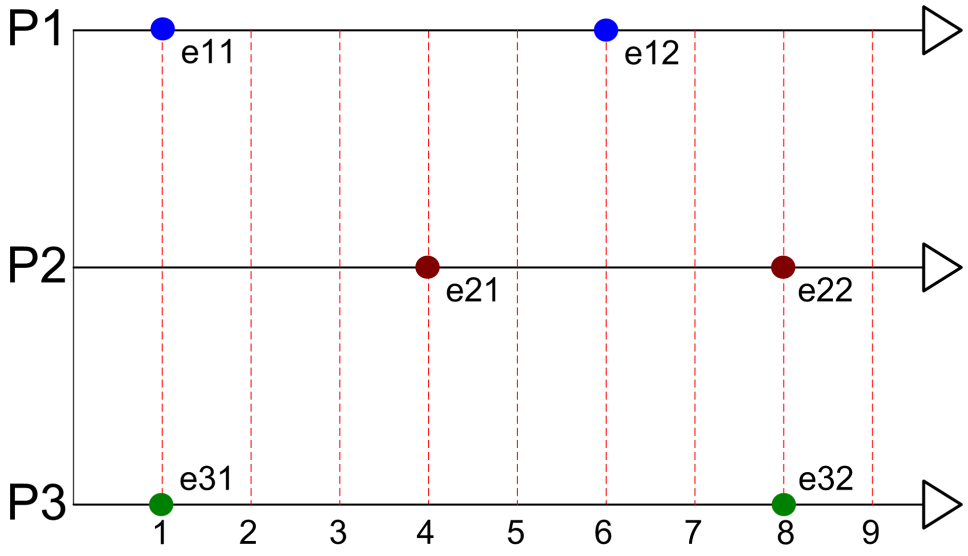

Figure 3.8: The approximation of the sequence of events.

Chapter 4

Related Work

4.1 Virtual Machine Creation and Management

VMware Inc. themselves offer many products around their virtualization technology. The most similar one to VAST is the VMware Lab Manager. It is automation software designed to enable users to create multi-tier virtual environments with custom network settings for software testing. It takes snapshots of the whole system in cases of test-failures, enables developers to easily share these snapshots to others and after a successful testrun tears down the infrastructure again.

Judging by its product description [VMw07] it is very similar to the environment construction part of VAST but offers additional functions which are rather related to providing virtual machines to the development staff rather than for automated testing.

What VMware Lab Manager lacks is the automation that VAST offers. It is possible to create virtual environments quickly and easily with it - but only manually. It seems that there is no programmatical access, or

something comparable, to make it work automatically and to be able to become part of a *continuous integration* cycle. Furthermore, unlike VAST, VMware Lab Manager doesn't offer any testing features itself except being able to integrate with 3rd party testing products - it is a solution only for creating and managing virtual environments.

There is another system, called SmartDomains [GPJ+08], which is also similar to the environment construction in VAST, but uses Xen virtual machines instead. It is based on SmartFrog as well and also uses the SmartFrog description language to define the configuration parameters of the virtual machines and where they should be created.

It is not intended to do testing, so it doesn't deploy a System Under Test and doesn't orchestrate tests - things which VAST does.

4.2 Automatic Distributed System Testing

I got the idea to use a state machine to run and verify tests out of the paper [UZCD99], though the authors call it *labeled transition system*. They suggest two testing architectures for distributed system testing: A global observer or distributed testers.

VAST is a mixture of both - the test controller being the global observer and the test runners being the distributed testers. The difference is that only the test controller is a point of control and observation, whereas in their paper every (distributed) tester can be a point of observation or a point of control and observation. The test runners of VAST are rather points of execution and observation, they are not in control.

Another approach similar to distributed system testing in VAST is described in the paper [TNS+99]. Their solution is designed to test specifically so-called distributed *Run Time Infrastructures*, so they are testing a runtime environment rather than the applications running on it. This already is a major difference, but still there are several similarities.

First of all they developed their own language to write test scripts, which is similar to the state machine description VAST uses. They require that the tests can be executed automatically, repeatedly, to enable regression testing, and that there are not special hooks into the Run Time Infrastructure required - which all matches VAST. They use RMI for the remote communication, too.

What distinguishes VAST from it is that there is a limit to the number of test runners - which they call *attachment point* - in their solution and that the development and storage of the test scripts differs from the SmartFrog approach.

Chapter 5

Conclusion

In this thesis I introduced a system called VAST - Virtual Environments for Automatic System Testing. Its purpose is to conduct automatic system tests in a distributed environment built out of virtual machines. It consists roughly of two parts: The environment construction, where the description of the system's infrastructure is interpreted and the virtual environment is built, and the system testing part, where test sequences are executed and verified using a reduced state machine.

5.1 Further Work

Due to the time limitations for this diploma thesis there are still things left undone.

VAST is designed to be an automatic test system and is therefore suited to be part of a continuous integration system. But in its current state it is not ready for that because it is lacking a simple interface to write tests with and the automatic construction of the SUT's packages are missing.

Right now VAST will halt when a test encounters an unexpected result, so developers get a chance to examine the systems to locate the error. But there might be the case where a failure is expected and one wants to know the failure percentage of a certain test. To measure this the test has to be run several times and in this case it is not desired that VAST stops. There should be an option which tells VAST to count the errors instead and to rerun the whole testsuite a defined number of times. Not halting when encountering errors is also desired when using VAST within *continuous integration* .

Another improvement concerns the state verification. As it is now, the test controller executes the actions of a test sequence and afterwards checks for the state's attributes. It doesn't take into account that states might be volatile and last only for a very short time - which might already have passed when the polling happens. Therefore a different verification mechanism should be implemented. Either the broadcast messages of attribute changes should be taken into account while executing a test sequence - so that the verification happens semi-concurrent - or another solution has to be found. A possiblity would be Anubis, an optional part of the SmartFrog component set:

> "Anubis is a fully distributed state monitoring and failure detection service designed to report distributed system states in a timely and consistent manner." [Mur05]

Anubis guarantees that published attributes - or states - are known by the whole system within a definied timespan. It also is able to detect (partial) stability of the system, which could be useful for the state verification. [Mur05]

Having a single test controller to orchestrate the tests, to verify the states and to report to the Avalanche server might prove to be a bottleneck in a sufficient large distributed system. An improvement would be to have more test controllers and divide the work between them. Their number should be according to the size of the distributed system and there should be an established command hierarchy between them. The top-most controller, which reports back to the Avalanche server, would then merely delegate the work to other controllers.

Another problem that needs to be addressed concerns the virtual machine templates. Right now it is necessary to copy the master images to the physical machines manually, before Avalanche is able to use them. This is rather a problem of Avalanche than of VAST, and there are already plans to realize some kind of software repository where the hosts ignited by Avalanche can download their required software components.

Bibliography

[Cru] http://cruisecontrol.sourceforge.net/. CruiseControl Website.

[FFSB04] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Design Patterns*. Head First. O'Reilly, 2nd edition, October 2004.

[GGL+03] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, and Peter Toft. Smartfrog: Configuration and automatic ignition of distributed applications. In *HP Openview University Association conference*. HP Labs, May 2003.

[GPJ+08] Xavier Grehant, Olivier Pernet, Sverre Jarp1, Isabelle Demeure, and Peter Toft. Xen management with smartfrog. In *Lecture Notes in Computer Science*, pages 205–213. Springer, 2008.

[Hud] https://hudson.dev.java.net/. Hudson: An extensible continuous integration engine.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*. Massachusetts Computer Associates, Inc., ACM, July 1978.

[Mah07] Helge Mahrt. Fixing and extending avalanche. Technical report, BA Stuttgart, September 2007.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[Mur05] Paul Murray. The anubis service. In *Hewlett-Packard Laboratories Technical Report*. HP Labs, June 2005.

[SG08] Ritu Sabharwal and Julio Guijarro. Avalanche: managing deployments for enterprise scale grids. In *Proceedings of the 1st Bangalore annual Compute conference*. HP Labs, ACM, 2008.

[Sma] http://wiki.smartfrog.org. SmartFrog Wiki Website.

[TNS+99] John Tufarolo, Jeff Nielsen, Susan Symington, Richard Weatherly, Annette Wilson, James Ivers, and Timothy C. Hyon. Automated distributed system testing: Designing an rti verification system. In *Proceedings of the 1999 Winter Simulation Conference*. ACM, 1999.

[UZCD99] Andreas W. Ulrich, Peter Zimmerer, and Gunther Chrobok-Diening. Test architectures for testing distributed systems, 1999.

[Val98] Antti Valmari. *The State Explosion Problem*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[VMw07] VMware Inc., 3401 Hillview Ave Palo Alto CA 94304 USA. *VMware Lab Manager*, 2007. Whitepaper.