Time Machine is used to automate Time Shift Testing on any resource in the enterprise, whether that resource is on-premises or in the cloud. Enterprises across the globe are using Time Machine with all the major cloud service providers, including Amazon Web Services (AWS). This document focuses on automating the provisioning process of AWS instances from pre-configured Amazon Machine Image (AMI) and then using those resources in automated time shift testing of an application. Some of these techniques highlight the unique situations that encompass extreme time shift testing such as not having the IP for the target system until the process has begun.

THE VIRTUAL CLOCK FOR APPLICATION TESTING

Time Machine provides software virtual clocks that enable you to time travel your applications into the future or the past, facilitating time shift testing on your date and time sensitive application logic, such as month-end, quarter-end, year-end processing, billing cycle, work flow, regulatory go live, and policy life cycle.

THE TIME MACHINE FLOATING LICENSE SERVER

Time Machine Floating License Server (TMFL) enables any instance of Time Machine to check out a license when started and return it to the license pool when stopped. Any Linux or Windows systems using Time Machine and configured properly can check out the license upon the start and stop of the daemons and services. The TMFL eliminates the need to apply a license key to each system in the deployment environment.

THE TIME MACHINE SYNC SERVER

Time Machine Sync Server (TMSS) allows Sync Groups to simultaneously broadcast virtual clocks to the entire test environment spanning multiple Time Machine systems regardless of time zone or geographic location. The TMSS contains a set of robust web services, which enable test automation and easy integration into test scripts or programs from any test tool.

TIME MACHINE AND AWS

There are many usage scenarios for leveraging Amazon Web Services or any cloud for that matter. If your specific use-case is one that involves starting, stopping and reusing the same instances for automating date and time sensitive tests, Time Machine could easily be a part of this scenario, just as it is in an on-premises deployment, with little or no alteration. Our example will deal with a more dynamic scenario where instances are provisioned and terminated on demand. A pre-requisite for this usage scenario is that an instance must be provisioned and built-out to specification of what the systems will need to look like when they are instantiated. In our example, we have installed Time Machine on Amazon Linux and added the licserverhost file to the configuration. This file points to the TMFL. Next, install and configure any infrastructure software that may be required such as Apache, MySQL, Git, Docker or JBoss, for example.

Once the instance is functional, shut it down and make an AMI from the stopped instance. This will serve as the template for future provisioning. Some other things needed to proceed are:

- The AWS CLI installed and configured where the scripts will be executed
- AWS SDK for Python. Otherwise known as Boto. These Python libraries wrap the AWS CLI.
- The AWS instance type. This reflects profile the instance will have allocated such as RAM and CPU and this will have a direct impact on cost.

- The AWS Security Group. These are essentially a list of port configurations.
- The SSH private key that will be used to access the instance.

PROVISION A NEW INSTANCE

The following example uses the AWS CLI with the Python wrappers to provision a new instance based on an AMI image.

```
def create_instance(path):
  ec2 = boto3.resource('ec2')
  inst1 = ec2.create_instances(ImageId='ami-d39b71ab',MinCount=1,
            MaxCount=1,
            SecurityGroups=['launch-wizard-3',],
            InstanceType='t2.micro',
            KeyName='sss')
```

where:

- *ami-d39b71ab* is the ID for the AMI image which is the template for the instance.
- *Launch-wizard-3* is the name for the security group chosen for the instance.
- *t2.micro* is the shape or size of the instance and specifies processor cores and RAM.
- *sss* is the name of the private key file generated with the ssh-keygen component of Secure Shell.

The query parameter requests the command return the ID for the instance created by this command.

GET THE PUBLIC IP ADDRESS

The following example uses the AWS CLI with the Python wrappers to get the public IP address for the instance provisioned in the previous step.

```
  newinstance = inst1[0].id
  base = ec2.instances.filter(InstanceIds=[newinstance,])
  for instance in base:
    print(instance.public_ip_address)
```

The query parameter requests the command return the public IP address for the instance created.

VALIDATING THE APPLICATION UNDER TEST

Time Machine and the Time Machine Sync Server are essential for enabling automation with time shift testing. The web services published by the TMSS are easily leveraged by any programming language or scripting tool that supports an API call. With our extreme time shift testing example, we use Selenium Web Driver to exercise the AUT. Our test case, from which this script derived, mandate that we execute this test with a specific date and time. The following example shows our adding of the target IP address that we created dynamically in a step above.

```
ws_data = {‘action':'ModifyTarget','name':'awsDemo','target_name':'aws',"target_address": self.__base_url}

ws_url = 'http://129.0.36.79:4600/tmSyncServer?'

r=requests.get(ws_url,params=ws_data)
```

where:

- *ModifyTarget* is the action or specific web service to which we supply the IP address of the new instance.
- *self.__base_url* is the variable containing the IP address of the new instance where the AUT is deployed..
- *ws_url* is the URL containing the path to the Time Machine Sync Server API, the default port is 4600.

In the next example, we enable the Sync Group defined in the TMSS for this AUT. Enabling the Sync Group enables the virtual clock specified in the Sync Group. We can also set the time here as well.

```
ws_data = {'username':'admin','password':'pwd1','action':'EnableSyncGroup','name':'awsDemo','abs_time':'021520202200','speed': '2'}

ws_url = 'http://129.0.36.79:4600/tmSyncServer?'

r=requests.get(ws_url,params=ws_data)
```

where:

- *EnableSyncGroup* is the action or specific web service called.
- *Abs_time* is the variable for absolute time. We set the time to 15-Feb-2020 at 10:00 PM local.
- *speed* is the variable for clock speed. We set the clock speed to be twice as fast as the system clock.

Lastly, we disable the Sync Group defined in the TMSS for this AUT. Disabling the Sync Group removes the virtual clocks from the target servers.

```
ws_data = {'username':'admin','password':'pwd1','action':'DisableSyncGroup','name':'awsDemo'}

ws_url = 'http://129.0.36.79:4600/tmSyncServer?'

r=requests.get(ws_url,params=ws_data)
```

where:

- *DisableSyncGroup* is the action or specific web service called.

TERMINATING THE INSTANCE

The following example uses the AWS CLI with the Python wrappers to terminate the instance.

```
ec2.instances.filter(InstanceIds=ids).terminate()
```

THE REST OF THE STORY

What happens between creating and terminating the instance is largely up to you. Many things are possible. Here is one example.

Continuous Deployment tools, such as Jenkins, Hudson, Bamboo, etc. are used to deploy a newly built application to the now running instance. The same process can start automated test scripts to run remotely on the deployment target with Pass/Fail results added to a Test Management system. Here is an example of notifications that use the collaboration tool Slack to communicate the results of the deployment and testing of the AUT:

```
jenkins APP [11:46 AM]
----------------
MasterJob_file_collector - #513 Started by changes from sroj (3 file(s) changed) (Open)

[11:46]
----------------
MasterJob_file_collector - #513 Success after 10 sec (Open)

[11:46]
----------------
deploy2Appservers - #94 Started by upstream project "MasterJob_file_collector" build number 513 (Open)

[11:46]
----------------
deploy2Appservers - #94 Success after 8.4 sec (Open)

[11:46]
----------------
UnitTests_selenium - #113 Started by upstream project "Regressiontests_script_collector" build number 191 (Open)

[11:47]
----------------
UnitTests_selenium - #113 Success after 35 sec (Open)
No Tests found.
```

Solution-Soft's Time Machine serves as the foundation for automating tests involving dates and times other than the current system clock. Add in the Time Machine Suite of Products and you have a powerful arsenal for Extreme Time Shift Testing. The TMFL allows for concurrent licensing of Time Machine; including enabling the newly provisioned instance in our example to check out a license dynamically and return the license to the pool upon its termination. The TMSS allows for dynamically setting or changing the virtual time for the application deployment nodes and its built-in API enables easy test automation from any program or test tool scripts.

SolutionSoft Systems, Inc.
2350 Mission College Blvd.
Suite #777
Santa Clara, CA 95054
U.S.A.

Toll Free: 1.888.884.7337
Phone: 1.408.346.1400
Fax: 1.408.346.1499

www.solution-soft.com

www.facebook.com/solutionsoft
www.twitter.com/solutionsoft
www.linkedin.com/solutionsoft