Using Test Clouds to Enable Continuous Integration Testing of Distributed Real-time and Embedded System Applications

Dr. James H. Hill
Dept. of Computer and Information Science
Indiana Univ.-Purdue Univ. Indianapolis
Indianapolis, IN, USA
hillj@cs.iupui.edu

Dr. Douglas C. Schmidt
Vanderbilt University
Nashville, TN
d.schmidt@vanderbilt.edu

ABSTRACT

It is critical to evaluate the quality-of-service (QoS) properties of enterprise distributed real-time and embedded (DRE) system early in the software lifecycle—instead of waiting until system integration time—to minimize the impact of rework needed to remedy QoS defects. Unfortunately, enterprise DRE system developers and testers often lack the necessary resources to support such testing efforts. This chapter discusses how test clouds (i.e., cloud-computing environments designed for testing) can provide the necessary testing resources. When combined with system execution modeling (SEM) tools, test clouds provide the necessary toolsets to perform QoS testing earlier in the software lifecycle. A case study of design and implementing resource management infrasutrcture from the domain of shipboard computing environments is used to show how SEM tools and test clouds can be used to identify defects in system QoS specifications and enforcement mechanisms earlier in the software lifecycle.

INTRODUCTION

Current trends and challenges. Enterprise distributed real-time and embedded (DRE) systems, such as large-scale traffic management systems, supervisory control and data analysis (SCADA) systems, and shipboard computing environments, are becoming increasingly ubiquitous. As these systems grow in scale and complexity they are becoming *ultra-large-scale* cyber-physical systems (Institute), which exhibit the following characteristics:

- Requirements for simultaneously satisfying competing and conflicting QoS properties, such as low latency, high reliability, and fault tolerance, in addition to meeting their functional requirements,

- Heterogeneous in both their operating environment (*e.g.*, target architecture and hardware resources) and technologies (*e.g.*, programming language and middleware), and

- Traditionally developed as monolithic, vertically integrated stove-pipes, which are brittle, and hard to implement, maintain, and evolve.

These and other related characteristics not only increase enterprise DRE system complexity, but also complicate their software lifecycle, resulting in elongated software lifecycles characterized by expensive project runs and missed delivery deadlines (Mann).

Due to the increase in complexity and size of enterprise DRE systems and their complicated software lifecycles, it is important to validate enterprise DRE system QoS properties early software lifecycle rather than waiting until complete system integration time (*i.e.*, late in the software lifecycle), when they are expensive and time-consuming to fix. System execution modeling (SEM) tools (Smith and Williams) are one method for enabling DRE system developers and testers to validate QoS properties during early phases of the software lifecycle. In particular, SEM tools provide enterprise DRE system developers and testers with the following capabilities:

- Rapidly model behavior and workload of the distributed system being developed, independent of its programming language or target environment, *e.g.*, the underlying networks, operating system(s), and middleware platform(s),

- Synthesize a customized test system from models, including representative source code for the behavior and workload models and project/workspace files necessary to build the test system in its target environment,

- Execute the synthesized test system on a representative target environment testbed to produce realistic empirical results at scale, and

- Analyze the test system's QoS in the context of domain-specific constraints, *e.g.* as scalability or end-to-end response time of synthesized test applications, to identify performance anti-patterns (Smith and Williams), which are common system design mistakes that degrade end-to-end QoS.

By using SEM tools to validate QoS properties throughout the software lifecycle, DRE system testers can locate and resolve QoS bottlenecks in a timely and cost-effective manner.

Although SEM tools facilitate early validation of enterprise DRE QoS properties, system testers may lack the necessary resources to support the testing efforts. For example, to support early integration testing of an enterprise DRE system, testers need hardware resources that may not be readily available until later in the software lifecycle. Likewise, both software and hardware resources may change throughout the software lifecycle. The number of test that must execute also often exceeds the operational capacity of resources available in-house for testing (Porter, Yilmaz and Memon). DRE system testers therefore need improved methods to support early integration testing of enterprise DRE system applications.

Solution approach → Use test clouds to support early integration testing. Cloud computing (Vouk) is an emerging computing paradigm where computing resources are managed by external service providers. End-users then provision (or request) the cloud's resources as needed to support their computational goals. Although cloud computing is primarily used for enterprise workloads, such as office applications, web servers, and data processing, the availability of the computing resources can also be provisioned to support testing efforts (*i.e.*, test clouds). For example, DRE system testers can provision resources available in the cloud to evaluate applications using resources that is not readily available in-house. Using test clouds is hard, however, without the proper infrastructure and tool support.

For example, test clouds are typically designed to support general-purpose workloads, which implies that cloud resources are allocated without any knowledge of how they will be used because testing is typically a domain-specific task. It is therefore the responsibility of end-users (*e.g.*, enterprise DRE system testers) to manage provisioned resources as follows:

- **Coordinate and synchronize testing efforts.** Different testing efforts require different testing methods. For example, when evaluating end-to-end response time, it is necessary to coordinate software running on many different machines for extended periods of time. It is also the responsibility of testers to supply the necessary framework to support such needs, if the test cloud does not provide it.

- **Gather data collected from many networked machines.** As the test is running, it generates metrics that can be used to analyze its behavior, *e.g.*, worst cast end-to-end response time for the system at different execution times. Unless the test cloud provides such capabilities, end-users are responsible for providing the necessary framework to support these needs.

- **Correlate and analyze collected metrics.** After metrics have been collected from many different networked machines, they must be correctly correlated to undergo analysis. Failure to correlate metrics can result in either false positive or false negative results. Unless the test cloud provides such capabilities, the end-users must do so themselves.

Completing the actions above can be a daunting task for enterprise DRE system developers and testers. Moreover, if not done correctly, each task must be repeated for each new enterprise DRE system that undergoes QoS testing within a test cloud. To simplify these tedious and error-prone tasks, this chapter shows by example how SEM tools can be extended to support test clouds, and overcome the challenges outlined above. In particular, this chapter describes how to:

- Create an instrumentation and logging framework to autonomously collect data from networked machines and store them in a central location while tests are executing.

- Create a test management framework to coordinate and synchronize testing efforts that execute on many networked hosts in a test cloud.

- Combine test clouds with continuous integration environments to execute integration tests that validate QoS properties in parallel with system development, with the goal of identifying and rectifying QoS-related defects during early phases of the software lifecycle.

BACKGROUND

Before beginning the discussion on using test clouds to evaluate enterprise DRE system QoS properties, we first examine the current state of testing with respect to cloud computing environments. In particular, *Testing as a Service (TaaS)* (Yu, Zhang and Xiang) is an emerging paradigm where testing processes, such as test data generation, test execution, and test analysis, is provided as a service to the end-user. This paradigm is opposed to always (re)inventing the test infrastructure for different application domains. Moreover, it tries to alleviate the overhead that is associated with testing, such as having dedicated human and computing resources.

Many companies now provide TaaS, *e.g.*, searching for TaaS on the Web locates a company named Sogeti (www.sogeti.com) that provides TaaS using a test cloud.[1] Their business model offers a pay-per-use software testing model that allows clients to perform both functional and performance testing on resources provisioned from their cloud. This model is similar to how the Amazon Cloud offers a pay-per-use model for their computing resources. In the Sogeti model, however, the services are test services that can be accessed via a web portal.

Lu et al. (Yu, Tsai and Chen) explored the feasibility of TaaS by deploying unit testing web service over a cloud. Their feasibility study highlighted the main concerns of deploying such a service, including (1) clustering requests (since many request and testing needs are different), (2) scheduling resources for testing (which is similar to traditional clustering scheduling problems), (3) monitoring testing resources and the test progress, and (4) managing processes in the overall cloud.

MOTIVATIONAL CASE STUDY: THE RESOURCE ALLOCATION CONTROL ENGINE (RACE)

The *Resource Allocation and Control Engine (RACE)* (Shankaran, Schmidt and Chen) is an open-source distributed resource manager system developed using the CIAO (Deng, Gill and Schmidt) implementation of the Lightweight CORBA Component Model (CCM) (Lightweight CORBA Component Model RFP) over the past decade. RACE deploys and manages Lightweight CCM application component assemblies (*i.e.*, called operational strings) based on resource availability/usage and QoS requirements of the managed operational strings. Figure 1 shows the architecture of RACE, which is composed of four components assemblies (*Input Adapte*r, *Plan Analyzer*, *Planner Manager*, and *Output Adapter*) that collaborate to manage operational strings for the target domain.

[1] The authors of this chapter have no affiliation with Sogeti.

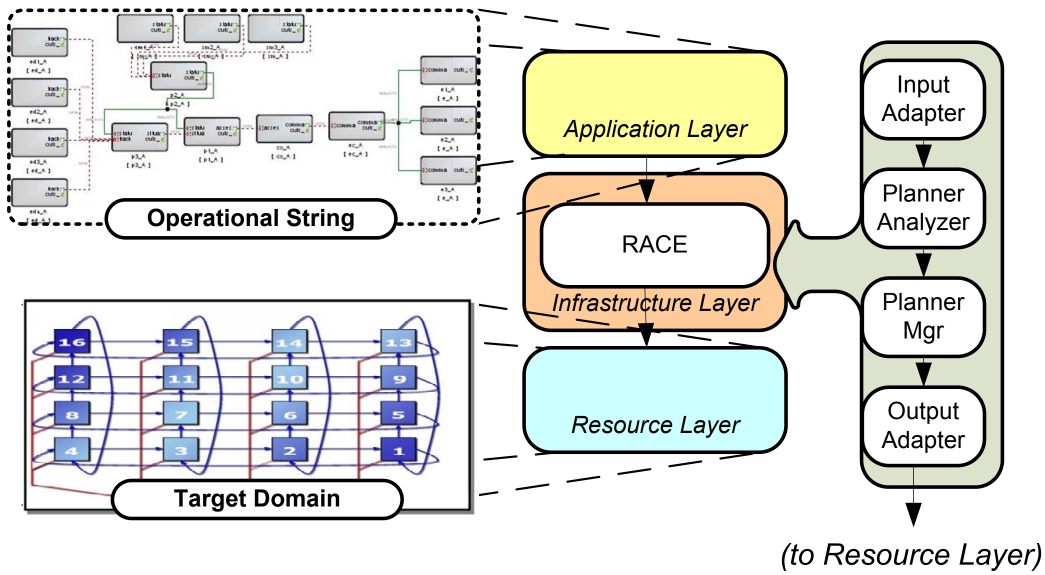

Figure. 1: High-level overview of the RACE architecture.

The initial implementation of RACE contained 26,644 lines of C++ code and 30 components. Subsequent enhancements to RACE (after 50 source code check-ins to the repository) added 14,041 lines of code and 9 components, for a total of 40,685 lines of code and 39 components.[2]

RACE performs two types of deployment strategies—*static* and *dynamic*—for enterprise DRE systems. Static deployments are operational strings created offline by humans or automated planners. RACE uses the information specified in a static deployment plan to map each component to its associated target host during the deployment phase of a DRE system. A benefit of RACE's static deployment strategy is its low runtime overhead since deployment decisions are made offline; a drawback is its lack of flexibility since deployment decisions cannot adapt to changes at runtime.

Dynamic deployments, in contrast, are operational strings generated online by humans or automated planners. In dynamic deployments, components are not given a target host. Instead, the initial deployment plan contains component metadata (*e.g.*, connections, CPU utilization, and network bandwidth) that RACE uses to map components to associated target hosts during the runtime phase of a DRE system. A benefit of RACE's dynamic deployment strategy is its flexibility since deployment decisions can adapt to runtime changes (*e.g.*, variation in resource availability); a drawback is its higher runtime overhead.

The RACE baseline scenario. The case study in this paper focuses on RACE's *baseline scenario*. This scenario exercises RACE's ability to evaluate resource availability (*e.g.*, CPU utilization and network bandwidth) with respect to environmental changes (*e.g.*,

[2] The number of lines of code in RACE was computed via SourceMonitor (www.campwoodsw.com/sourcemonitor.html) and the number of components in RACE was counted manually.

node failure/recovery). Moreover, it evaluates RACE's ability to ensure lifetime of higher importance operational strings deployed dynamically is greater than or equal to the lifetime of lesser importance operational strings deployed statically based on resource availability.

Since RACE performs complex distributed resource management services we wanted to evaluate RACE's QoS properties as early as possible in the software development lifecycle. We believed this would help us overcome the serialized-phased development problem (Rittel and Webber) where the system is developed in different phases and tested functional, but QoS evaluation does not occur until complete system integration time. In particular, we wanted to know as early as possible if RACE could evaluate resource availability with respect to environmental changes to properly manage operational strings deployed dynamically versus those deployed statically.

Because RACE is considered infrastructure components, we needed application components to successfully evaluate its baseline scenario. Due to serialized-phasing development, applications would not exist for several months after development started. We therefore relied on CUTS (see Sidebar 1) to provide the application-level components for evaluating RACE's baseline scenario.

Sidebar 1. The CUTS System Execution Modeling Tool

CUTS (Hill, Edmondson and Gokhale) is a next-generation system execution modeling (SEM) tool that enables DRE system developers and testers to performance system integration tests that valid QoS properties during early phases of the software lifecycle. This is opposed to waiting to complete system integration time to perform such evaluation, which can be too late to resolve problems in a timely and cost-effective manner. DRE system developers and testers use CUTS via the following steps:

1. Use domain-specific modeling languages (Ledeczi, Bakay and Maroti) to model behavior and workload at high-levels of abstraction (Hill, Tambe and Gokhale, Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems);
2. Use code generation techniques (Hill and Gokhale, Using Generative Programming to Enhance Reuse in Visitor Pattern-based DSML Model Interpreters) to synthesize a complete test systems from constructed models that conform to the target architecture (*i.e.*, the generate components look and feel like the real component in terms of their exposed attributes and interfaces); and
3. Use emulation techniques (Hill, Slaby and Baker) to execute the synthesized system on its target architecture and validate its QoS properties in its target execution environment.

DRE system developers and testers can also replace emulated portions of the system with its real counterpart as its development is completed. This allows DRE system testers to perform continuous system integration testing, *i.e.*, the process of execution system integration test to validate QoS properties continuously throughout the software lifecycle.

Although we used CUTS to provide the non-existing application-level components, we did not have enough physical resources in-house to conduct the desired testing efforts.

We therefore decided to use a cloud environment for evaluating RACE's baseline scenario. In particular, we leveraged a test cloud powered by Emulab software (Ricci, Alfred and Lepreau) to support our testing efforts. We selected Emulab because it allowed us to provision both host and network resources to create network topologies that emulate production environments. This capability enabled us to create realistic experiments based on realistic networking conditions (*i.e.*, we were not bound to the *predefined* network topology and configuration of emerging TaaS paradigms built atop a test cloud).

Like many clouds, Emulab provides the infrastructure support for managing its resources, but it does not provide readily available infrastructure that supports testing DRE systems. It was therefore our responsibility to develop the necessary infrastructure that allowed us to (1) coordinate and synchronize testing efforts, (2) gather data collected from many different nodes, and (3) execute many tests automatically.

Since we were developing RACE, we could have easily created simple shell scripts (*i.e.*, the traditional way of manage the test process of a DRE system) designed for RACE to manage the testing process in Emulab. This approach, however, would not be an ideal solution since we reduce the mobility of our solution. Moreover, it would be hard for us to leverage much of Emulab's existing infrastructure (*e.g.*, configuration and dynamic resource management) that requires stateful and interactive processes.

The remainder of this chapter therefore discusses how we extended the CUTS SEM tool to support the Emulab test cloud. The goal of this extension is to provide the underling infrastructure above and beyond what Emulab provides so that testing DRE systems is not only an easy process, but also an automated one.

COMBINING SYSTEM EXECTION MODELING TOOLS AND CLOUD COMPUTING ENVIRONMENTS

This section discusses the extension we added to the CUTS SEM tool to support evaluating RACE in the Emulab test cloud. In particular, we describe the infrastructure and logging infrastructure implemented to collect data for analyzing the RACE baseline scenario. We also present the extensions added to CUTS to support managing testing exercises in Emulab and show how we integrated the test management extensions and the logging/instrumentation extensions to create a complete framework for executing many test in the Emulab test cloud.

Instrumentation and Logging Infrastructure for Test Clouds

Enterprise DRE systems consist of many software components executing on many hosts that are connected via a network. When validating their QoS properties, *e.g.*, end-to-end response time, scalability, and throughput, it is necessary to collect data about the systems behavior in the target environment. Such behaviors could be execution lifecycle events, the state of the system at different points in time, or data points needed to calculate the end-to-end response of an event.

Data (or metrics) in a distributed environment, such as a test cloud, is collected and analyzed either *offline* or *online*. In offline collection and analysis, data from each host is written to local persistent storage, *e.g.*, a file, while the system executes in the test cloud. After the system is shutdown, the collected data in local storage on each host is combined and analyzed. The advantage of this approach is network traffic is kept to a minimum since collected data is not transmitted over the network until after the system is shutdown. The disadvantage of this approach is collected data is not processed until the system is shutdown, which can pose a problem for enterprise DRE systems with a long execution lifetime, or when trying to monitor and analyze a system in real-time. More importantly, once a system has shutdown, its resources are released back into the cloud, so there is a potential risk that data stored on local storage can be lost if it is not completely removed.

In online collection and analysis, data is collected and transmitted via network to a central host. The advantage of this approach is that it allows metric analysis to occur on a host that has little or no impact on system performance. The disadvantage of online collection and analysis is that it is necessary to devise a strategy for efficiently collecting data and submitting it to a central location without negatively impacting the executing systems QoS – especially if the system generates heavy network traffic. Although online collection has it disadvantages, using online data collection a test cloud is more practical because it guarantees metrics are collected and archived periodically during execution (*i.e.*, before resources are given back to the cloud). Moreover, online data collection can facilitate real-time feedback.

Extending CUTS with logging and instrumentation infrastructure. Instrumentation and logging infrastructure should not be (re-)implemented for each application undergoing testing in a test cloud. Moreover, it should not be a something that DRE system and developers must provide. Instead, it should be a service provided by the test cloud that DRE systems use to collect and archive metrics of interest. This way, DRE system testers and developers can focus on deciding what to collect and let the test cloud infrastructure determine the best method for collecting metrics from the DRE system under test.

Figure 2 gives an overview of the how CUTS was extended with logging and instrumentation infrastructure to support the Emulab test cloud. As shown in this Figure, each host provisioned in the test cloud executes a logging client. Likewise, the controller node for the provisioned assets executes a single logging server. The controller node for the provisioned assets is not the same as the controller node for the cloud because each experiment executing in a test cloud has a controller node responsible for coordinating task on the different test nodes as described in the section titled Managing DRE System Tests in the Cloud.

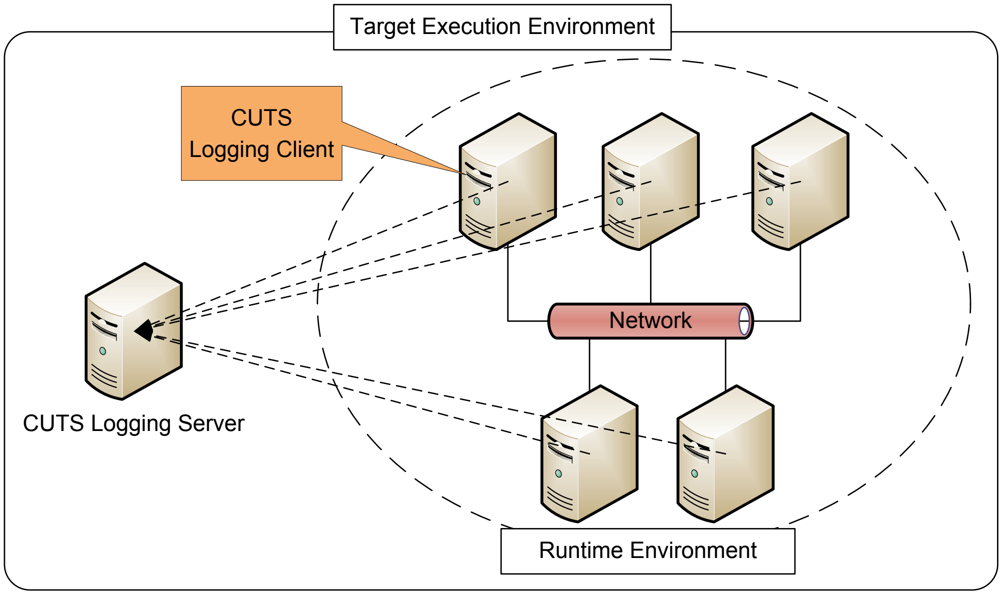

Figure 2. Overview of CUTS logging and instrumentation framework to support the Emulab test clouds.

The logging client is responsible for collecting data from individual software components executing on its host. As the logging client receives data from software components, it submits it to the logging server for storage and analysis. The logging client does not know the physical location (*i.e.*, IP address) of the logging server because it can change depending on what assets are provisioned by the test cloud for a given experiment. Instead, the logging client knows the test-specific hostname of the logging server and its listening port. Likewise, we can further loosen the coupling between the logging client and logging server by using a naming service to resolve the location of the logging server (*i.e.*, request the logging server for a given test).

In contrast to the logging client resolving the logging server, the process software components use to resolve the logging client is straightforward. Since software components submit metrics to logging clients executing on their host machine, the software components only need to know what port the logging client is listening. After a software component opens a connection to the logging client's port, it is able to submit data for collection and archiving.

On data formatting and archiving. Since a DRE system is distributed in nature—and its metrics can range from structured to unstructured data and binary to text data—there are many way to store data. We therefore extended CUTS to support both a text and binary methods for archiving metrics. In the text method, metrics are archived in string format—similar to log files of an execution trace stored in a database.

```
void Input_Adapter_Component::handle_input (Input_Event * ev) {
  CUTS_CLIENT_LOGGER->log ("begin processing input %s at %d",
                           ev->input_name (),
                           ACE_OS::gettimeofday ().msec ());
```

```
  // make call to planner manager component

  CUTS_CLIENT_LOGGER->log ("ending processing input %s at %d",
                           ev->input_name (),
                           ACE_OS::gettimeofday ().msec ());
}
```

Listing 1. Example source code for collecting software data from a software component using text formatting.

Listing 1 shows how CUTS collects text-based data from the Input Adapter component in RACE. The advantage of this approach is that data values can be viewed just by viewing the contents of the archive (*i.e.*, the database where metrics are stored). The disadvantage of this approach is that all metrics must be converted to string format, and vice versa, which can result in unwanted overhead—especially when collecting metrics from real-time software components.

In the binary method, data is stored in its raw binary representation. The advantage of this approach is that there is no extra overhead associated with packaging the data. The disadvantage of this approach is that special care needs to be taken when packing and unpacking the data if the endian-ness of the host used for data analysis is different than the host where the data originated. Likewise, it is almost impossible to see the contents of the data just by viewing the archive since the data will appear to be *encrypted.* This inherent characteristic, however, can add a layer of security to the data when many test execute in the same test cloud.

From our experience, the text-based format for collecting and storing metrics is easier to integrate into an existing application than the binary method when using intrusive data collection (*i.e.*, modifying the source code to collect metrics of interest). This simplicity occurs because the binary format method requires a special framework to manage packaging and unpackage the data, which can add unwanted complexity to the solution. In contrast, the text-based format, can be achieved using simple string-formatting semantics as shown in Listing 1.

When performing non-intrusive data collection (*i.e.*, collecting data without modifying the original source code) within the test cloud, then either binary- or text-based data collection suffices because the collection mechanisms are hidden from the client. For example, many DRE systems will use a logging framework for collecting text-based log messages. It is therefore possible to intercept those messages and pass them to the CUTS logging server—assuming the log messages contain metrics of interest.

Framework: log4j
Language: Java
How to integrate: Update log4j.properties (or similar file)
Illustrative example:

```
# define the loggers
```

```
log4j.rootCategory=ALL, C, A

# console appender
log4j.appender.A=org.apache.log4j.ConsoleAppender
log4j.appender.A.layout=org.apache.log4j.PatternLayout
log4j.appender.A.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

# CUTS appender
log4j.appender.B=cuts.log4j.LoggingClientAppender
log4j.appender.B.LoggerClient=corbaloc:iiop:localhost:20000/LoggingClie
nt
```

Framework: ACE Logging Facilities
Language: C++
How to integrate: Update svc.conf (or similar file)
Illustrative example:

```
dynamic CUTS_ACE_Log_Interceptor Service_Object * \
  CUTS_ACE_Log_Interceptor:_make_CUTS_ACE_Log_Interceptor() active \
  "--client=corbaloc:iiop:localhost:20000/LoggingClient"
```

Listing 2. Using interceptors to non-intrusively collect log messages in the log4j and ACE logging framework.

Listing 2 shows how CUTS can intercept log messages of the ACE logging framework (Schmidt), which is a C++ logging framework used in many enterprise DRE systems. The listing also shows how CUTS can intercept log message of the log4j logging framework (logging.apache.org), which is a Java framework used by many existing Java applications. As shown in this listing, the configuration files define what interceptors to load. When the CUTS interceptors are loaded into their respective framework, the interceptor receives log messages and forwards them to the CUTS Logging Server.

For the binary case, it is possible to use dynamic binary instrumentation tools to re-write the binary program and inject instrumentation points into the program as it executes. Similar to the logging framework interceptor discussed above, the callback methods collect data of interest and forward it to the CUTS logging server.

Summary. Whether using text- or binary-based formatting, or intrusive vs. non-intrusive, when collecting metrics in a test cloud, the approach must seamlessly extract metrics from each host and store them in a central location. Failure to do so can result in metrics being lost after provisioned resources are released back to the test cloud. Since the Emulab test cloud does not provide such functionality out-of-the-box, we extended the CUTS SEM tool with a logging and instrumentation framework to support our testing efforts of RACE in the Emulab test cloud.

Managing DRE System Tests in the Cloud

Enterprise DRE system testing and execution environments, such as the Emulab test cloud, consist of many nodes. Each node in the target environment is responsible for executing many different processes that must support the overall goals of the execution environment. For example, nodes in the execution environment may host processes that manage software components, processes that synchronize clocks with a central server, and processes that collect instrumentation data.

When dealing with many processes that must execute on a given node, the traditional testing approach is to create a script that launches all processes for that given node. For example, in Unix environments, Bash scripts can be used to launch a set of processes. The advantage of using such scripts is that it provides a *lightweight and repeatable* approach for ensuring that all process for a given node are launched, and done so in the correct order. Moreover, the script can be easily updated to either add or remove processes as needed.

Although scripts are sufficient when working in a distributed testing environment, its approach adds more complexity to the testing process in a test cloud. First, scripts are only repeatable after *manually* invoking the script. This means that enterprise DRE system developers much manually connect to each host in the test cloud to (re)start the script. If it is possible to remotely connect to the host via automation to (re)start the script, then it is *hard* to know what processes are associated with a given script if multiple scripts start similar processes.

Secondly, such scripts do no easily support injection/removal of processes on the fly. It is assumed in the script used to configure the host before the test is executed. At that point, all processes injected/removed from the host are assumed to be part of all the processes on that machine. Lastly, it is hard to switch between different execution environments where an execution environment is set of environment variables and a set of processes that are needed for a test to execute correctly in the test cloud.

Extending CUTS with test management infrastructure. Because of the shortcomings of existing approaches for managing DRE system tests, which are used heavily in the Emulab test cloud, we extended CUTS with test management infrastructure. More specifically, we added the following entities to CUTS:

The CUTS Node Manager is a daemon that manages processes executing on its respective node, similar to a traditional task manager. The difference between a traditional task manager and the CUTS Node Manager is that the CUTS Node Manager exposes an interface that is used by the Node Manager Client (explained next) to remotely spawn new processes on the node, or terminate existing processes executing on the node.

The CUTS Node Manager also provides mechanisms for remotely starting all process under its control without having to physically restart the machine in the test cloud. The CUTS Node Manager accomplishes this using a concept called *virtual environments,* which is a set of environment variables and processes that define a self-contained execution environment for testing purposes. Listing 3 shows an example configuration for the CUTS Node Manager.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<cuts:node>
  <environment id="RACE.HEAD" inherit="true" active="true">
    <!-- environment variables for this environment -->
    <variable name="NameService"
        value=">${TAO_ROOT}/orbsvcs/Naming_Service/Naming_Service" />
    <variable name="RACE_ROOT"
        value=">/opts/RACE" />

    <startup>
      <!-- NamingService -->
      <process id=”dance.naming.service”>
        <executable>${NameService}</executable>
        <arguments>-m 0 –ORBEndpoint
          iiop://localhost:60003 -o ns.ior</arguments>
      </process>

      <!-- DAnCE node manager -->
      <process id="dance.nodemanager.pingnode">
        <executable>${DANCE_ROOT}/bin/dance_node_manager</executable>
        <arguments>-ORBEndpoint iiop://localhost:30000
          -s ${DANCE_ROOT}/bin/dance_locality_manager
          -n PingNode=PingNode.ior -t 30
          --instance-nc corbaloc:rir:/NameService</arguments>
        <workingdirectory>../lib</workingdirectory>
      </process>

      <!-- //... -->
    </startup>
  </environment>

  <environment id="RACE.Baseline" inherit="true" active="false">
    <!-- environment variables for this environment -->
    <variable name="NameService"
        value=">${TAO_ROOT}/orbsvcs/Naming_Service/Naming_Service" />
    <variable name="RACE_ROOT"
        value=">/opts/RACE-baseline" />

    <startup>
      <!-- NamingService -->
      <process id=”dance.naming.service”>
        <executable>${NameService}</executable>
        <arguments>-m 0 –ORBEndpoint
          iiop://localhost:60003 -o ns.ior</arguments>
      </process>

      <!-- DAnCE node manager -->
      <process id="dance.nodemanager.node1">
        <executable>${DANCE_ROOT}/bin/dance_node_manager</executable>
        <arguments>-ORBEndpoint iiop://localhost:30000
          -s ${DANCE_ROOT}/bin/dance_locality_manager
          -n Node1=Node1.ior -t 30
          --instance-nc corbaloc:rir:/NameService</arguments>
        <workingdirectory>../lib</workingdirectory>
      </process>
```

```
      <!-- //... -->
    </startup>
  </environment>
</cuts:node>
```

Listing 3. Example configuration for the CUTS Node Manager illustrating the usage of virtual environments to manage different execution configurations.

As shown in this listing, there are 2 different virtual environments defined in the configuration where the main difference is the value of the variable named `RACE_ROOT`. The environment named `RACE.HEAD` is the active virtual environment when the CUTS Node Daemon is first launched. At runtime, however, it is possible to switch between the `RACE.HEAD` and `RACE.Baseline` virtual environments.

This runtime flexibility is beneficial when testing many different configurations that require different execution environments, *e.g.*, different versions of RACE that have different software dependencies, because DRE system developers and testers do not have to *physically* reset (or restart) the machine to switch configurations. Moreover, DRE system developers and testers do not have to *manually* terminate existing process and execute a different script just to switch configurations. Instead, the CUTS Node Manager can manage all the processes for each execution environment, and can quickly swap between them.

The CUTS Node Manager Client is an application that allows end-users to remotely control a CUTS Node Manager. The CUTS Node Manager Client therefore prevents end-users from having to manually log into each machine in the test cloud to modify its configuration. For example, DRE system developers and testers can remotely switch between different virtual environments, spawn new processes on remote host, or terminate an existing process on a remote host. The CUTS Node Manager Client is mainly used by the CUTS Test Manager (explained next) to inject behaviors into the execution environment at runtime. For example, the CUTS Test Manager can use to CUTS Node Manager Client to terminate a remote process hosting software components to evaluate the robustness of the system under testing during faults.

The CUTS Test Manager is an application that executes for a user-specified amount of time and is responsible for managing the testing exercise of a DRE system in the test cloud. The CUTS Test Manager application achieves this by first, wrapping itself around the deployment tools for the system under test. For example, when evaluating the RACE baseline scenario, the CUTS Test Manager wraps itself around DAnCE, which is a deployment and configuration engine for CORBA Component Model (CCM) applications. Listing 4 highlights a portion of the CUTS Test Manager configuration that defines what tool to use to deploy the system under testing into the test cloud.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<cuts:test>
  <startup>
    <executable>${DANCE_ROOT}/bin/dance_plan_launcher</executable>
```

```
    <arguments>-x RACE.cdp -k file://EM.ior</arguments>
  </startup>

  <shutdown>
    <executable>${DANCE_ROOT}/bin/dance_plan_launcher</executable>
    <arguments>-x RACE.cdp -k file://EM.ior -s</arguments>
  </shutdown>

  <!-- // remainder of test script -->
</cuts:test>
```

Listing 4. Snippet of the CUTS Test Manager configuration illustrating the use of DAnCE to deploy RACE into the test cloud.

As shown in the listing above, the CUTS Test Manager uses the `<startup>` section to determine how to launch the system under test (*i.e.*, RACE in the example above). The `<shutdown>` section determines how to stop the current system under testing. Because the startup and shutdown sections are generic, the CUTS Test Manager can wrap any deployment tool.

While the CUTS Test Manager is active (*i.e.*, the test is executing in the test cloud), the CUTS Test Manager can execute test actions. A *test action* is an operation that occurs independently of the system under test, but can have an effect on the system under test. For example, the CUTS Test Manager can use the CUTS Node Manager Client to send a command to the CUTS Node Manager to terminate an existing process on the machine.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<cuts:test>
  <!-- // startup/shutdown commands removed from script -->

  <actions>
    <action delay='30' waitforcompletion='true'>
      <executable>${CUTS_ROOT}/bin/cutsnode</executable>
      <arguments>-ORBInitRef
        NodeManager=node1.isislab.vanderbilt.edu:50000/CUTS/NodeManager
        --reset</arguments>
    </action>

    <!-- // more test actions go here -->
  </actions>
</cuts:test>
```

Listing 5. Snippet of the CUTS Test Manager configuration illustrating the use of test actions to alter the behavior to the test environment at runtime.

Listing 5 shows a snippet of the CUTS Test Manager configuration for sending commands to nodes throughout the test execution. As shown in this listing, the CUTS Test Manager sleeps for 30 seconds. After the sleep delay, it resets the CUTS Node Manager running on `node1.isislab.vanderbilt.edu` (i.e., forces all its managed processes to restart).

Summary. When evaluating an enterprise DRE system in a test cloud, such as Emulab, it is necessary for the test cloud to have infrastructure support for managing the complete testing operation. As discussed in this section, this support mainly involves having infrastructure manage processes executing on remote machines (*i.e.*, the CUTS Node Manager and CUTS Node Manager Client) and infrastructure for managing different test scenarios (*i.e.*, the CUTS Test Manager). Without such infrastructure in place, it is hard for DRE system testers to leverage a test cloud to for their testing exercises because they will spend significant time and effort trying to manage many remote resources, which is an daunting task.

Coordinating Logging, Instrumentation, and Testing Management Infrastructure in the Test Cloud

The previous two sections discussed topics related to (1) logging and instrumentation infrastructure for collecting test data from a DRE system and (2) test management infrastructure for coordinate the testing effort across different nodes in the test cloud. The discussion in each section and resulting artifacts, however, are disjoint in that the logging and instrumentation infrastructure does not know about the test management infrastructure.

A disjoint solution has both its advantages and disadvantages. For example, one advantage of a disjoint solution is that the loose coupling between the two allows either one to be used without the other. For example, it is possible to use the CUTS Node Manager to manage virtual environments and processes without including the CUTS Test Manager. Likewise, it is possible to use the CUTS Test Manager to manage different test scenarios without ever using the CUTS Logging Client and CUTS Logging Server.

One disadvantage of having a disjoint solution is that it is hard to ensure that data collected by the logging and instrumentation infrastructure is associated with the correct test. This association is not problematic if only one enterprise DRE system is instrumented for a given test. If there are multiple systems being instrumented that use the same logging infrastructure and want their data associated with different tests, however, then having a disjoint solution is not ideal. The problem is that there is no easy way to ensure the test data is associated with the correct test without introducing some form of compiling that permits data to be associated with a given test.

Integrating logging and test management infrastructure in CUTS. Since there is occasional need associate collected data with a given test, we allow the CUTS Test Manger to support testing services. A *testing service* is an entity that adds domain-specific behavior based on the lifetime of the testing exercise. The main motivation for supporting testing services is that it is hard to know in advance all the needs of a testing exercise that must be associated with the testing lifecycle. For example, some testing exercises may need to associate data with a given test, and other testing exercises may not. Testing services therefore will allow both needs to be supported without compromising any needs, or the flexibility of CUTS extensions for testing DRE systems in test clouds.

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<cuts:test>
  <!-- // startup/shutdown commands and test actions removed -->
  <services>
    <service id="daemon">
      <location>CUTS_Testing_Server</location>
      <entryPoint>_make_CUTS_Testing_Server</entryPoint>
      <params>-ORBEndpoint iiop://localhost:50000</params>
    </service>

    <service id="logging">
      <location>CUTS_Testing_Log_Message_Listener</location>
      <entryPoint>_make_CUTS_Testing_Log_Message_Listener</entryPoint>
      <params>-ORBInitRef
       LoggingServer=
         corbaloc:iiop:localhost:20000/LoggingServer</params>
    </service>
  </services>
</cuts:test>
```

Listing 6. Snippet of CUTS Test Manager configuration illustrating services to be loaded into the test manager.

Listing 6 shows a portion of the CUTS Test Manager configuration that loads the CUTS Logging Server as a service (*i.e.*, logging in Listing 6) into the CUTS Test Manager. It also loads a service that exposes an external endpoint (*i.e.*, daemon in Listing 6) for remotely connecting to the CUTS Test Manager. Because the CUTS Logging Server is loaded in the CUTS Test Manager, it has access to the CUTS Test Manger attributes, such as test id and test lifecycle events (*e.g.*, start, stop, and pause).

The CUTS Testing Service also exposes an interface that allows clients to query information about the current test, such as test id. When the DRE system is deployed into the test cloud, the loggers use the interface provided by the CUTS Test Manager to query for a new test id. The loggers then associated the returned test id with the data that it submits to the CUTS Logging Client and CUTS Logging Server.

Summary. Although separating the logging and instrumentation infrastructure from the test management infrastructure has advantages, there are cases when the two need should be coupled. Infrastructure for the test cloud should therefore support both needs. Otherwise, it can be hard to associate data with a given test.

One concern not discussed in this section is when new software components are deployed during the middle of a test. If there is only one test executing, then the solution is trivial (*i.e.*, request the id of the currently executing test). If there are multiple tests executing (*i.e.*, multiple test ids), then it can be hard to determine what test the new components should be associated with. One solution to address this problem is to pre-assign test ids before the system is deployed into the test cloud. This way, when new components come online they know which test to associate its data. Otherwise, if you rely on auto-generated test ids (as discussed above), then you need some mechanism in place that can determine which groups of test ids belong to the same overarching test. This, however, can be *hard* to do if the testing exercise is dynamic (*i.e.*, has many software components that are deployed and destroyed throughout the testing lifecycle).

EXPERIMENTAL RESULTS: USING TEST CLOUDS TO EVALUATE THE RACE BASELINE SCENARIO

This section shows the design and results of experiments that applied the extended version of CUTS, which we call CiCUTS from henceforth, to evaluate the RACE's baseline scenario in the Emulab test cloud. These experiments evaluated the following hypotheses:

- **H1:** CiCUTS allows developers to understand the behavior and performance of infrastructure-level applications, such as RACE, within a test cloud, such as Emulab, before system integration; and
- **H2:** CiCUTS allows developers use test clouds for continuously evaluating QoS properties of infrastructure-level applications throughout its development lifecycle.

Experiment Design

To evaluate the two hypotheses in the context of the RACE baseline scenario, we constructed 10 operational strings using CUTS. Each string was composed of the same components and port connections, but had different importance values and resource requirements to reflect varying resource requirements and functional importance between operational strings that accomplish similar tasks, such as a primary and secondary tracking operation. Figure 3 shows a PICML[3] model for one of the baseline scenario's operational strings consisting of 15 interconnected components represented by the rounded boxes. This operational string was replicated 10 times to create the 10 operational strings in the baseline scenario.

[3] The Platform Independent Component Modeling Language (PICML) (Gokhale, Balasubramanian and Balasubramanian) is a domain-specific modeling language for modeling compositions, deployments, and configurations of Lightweight CCM applications.

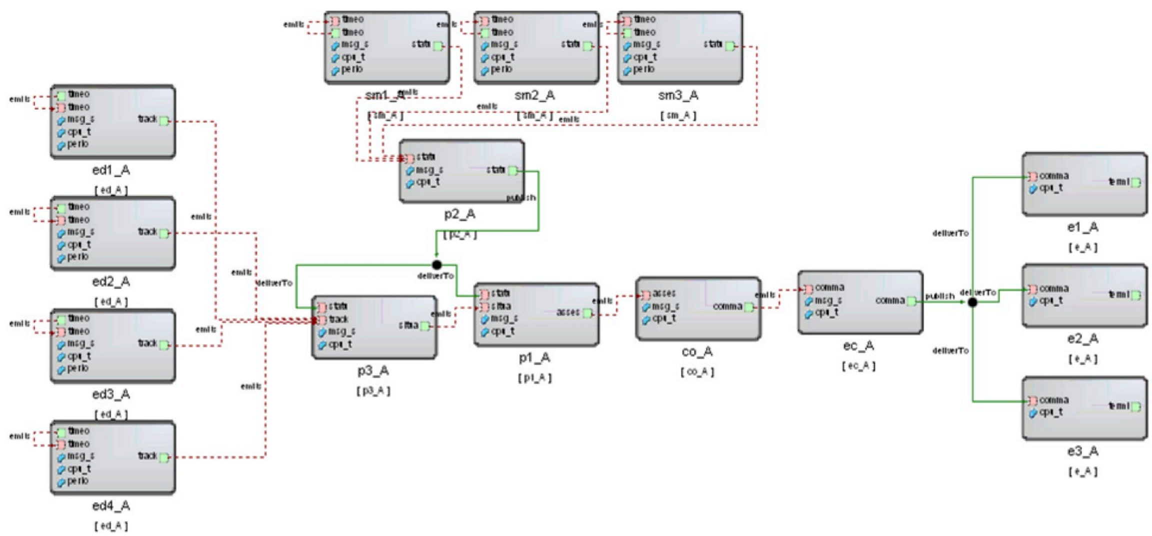

Figure 3. Graphical model of the replicated operational string for the baseline scenario.

The four components on the left side of the operational string in Figure 3 are sensor components that monitor environment activities, such as tracking objects of importance using radar. The four components in the top-middle of Figure 3 are system observation components that monitor the state of the system. The four linear components in the bottom-center of Figure 3 are planner components that receive information from both the system observation and sensor components and analyze the data, *e.g.*, determine if the object(s) detected by the sensor components are of importance and how to (re)configure the system to react to the detected object(s). The planner components then send their analysis results to the three components on the right side of Figure 3, which are effector components that react as stated by the planner components, *e.g.*, start recording observed data.

To prepare RACE's baseline scenario for CiCUTS usage, we used PICML to construct the 10 operational strings described above. We then used the CUTS to generate Lightweight CCM compliant emulation code that represented each component in the operational string managed by RACE (see Figure 3) in the baseline scenario. We also used PICML to generate the operational strings' deployment and configuration descriptors for RACE.

Table 1. The importance values of the baseline scenario operational strings.

Operation String	Importance Value
A – H	90
I – J	2

The deployment of each operational string used the strategy specified in Table 1. The importance values[4] assigned to each operational string reflects its mission-critical ranking

[4] These values are not OS priorities; instead, they are values that specify the significance of operational strings to each other

with respect to other operational strings. We chose extreme importance values because RACE was in its initial stages of development and we wanted to ensure that it honored importance values when managing operational strings. It is worth noting that we can easily change the importance values to evaluate RACE's ability to handle operational strings with closely related importance values. This, however, was outside the scope of our experiments. Finally, we annotated RACE's source code with the logging messages so that the CUTS Logging Client and CUTS Logging Server can collect data about the test, such as time of operational string deployment/teardown or time of node failure recognition.

To run the experiments using CiCUTS, we create configuration scripts for the CUTS Test Manager that captured the serialized flow of each experiment. The configuration scripts contained commands that (1) signaled RACE to deploy/teardown operational strings, (2) sent commands to individual nodes to cause environmental changes, and (3) queried the logging database for test results. Finally, we created a custom graphical display that analyzed the log messages to show whether the lifetime of dynamic deployments exceed the lifetime of static deployments based on resource availability with respect to environmental changes.

Experiment Results

This section presents the results of experiments that validate H1 and H2 in context of the RACE baseline scenario.

Using CiCUTS to understand the behavior and performance of infrastructure-level applications when testing in a test cloud. H1 conjectured that CiCUTS assists in understanding the behavior and performance of infrastructure-level applications, such as RACE, using a test cloud well before system integration. Figure 4 shows an example result set for the RACE baseline scenario (*i.e.*, measuring the lifetime of operational strings deployed dynamically vs. operational strings deployed statically) where 2 hosts were taken offline to simulate a node failure.

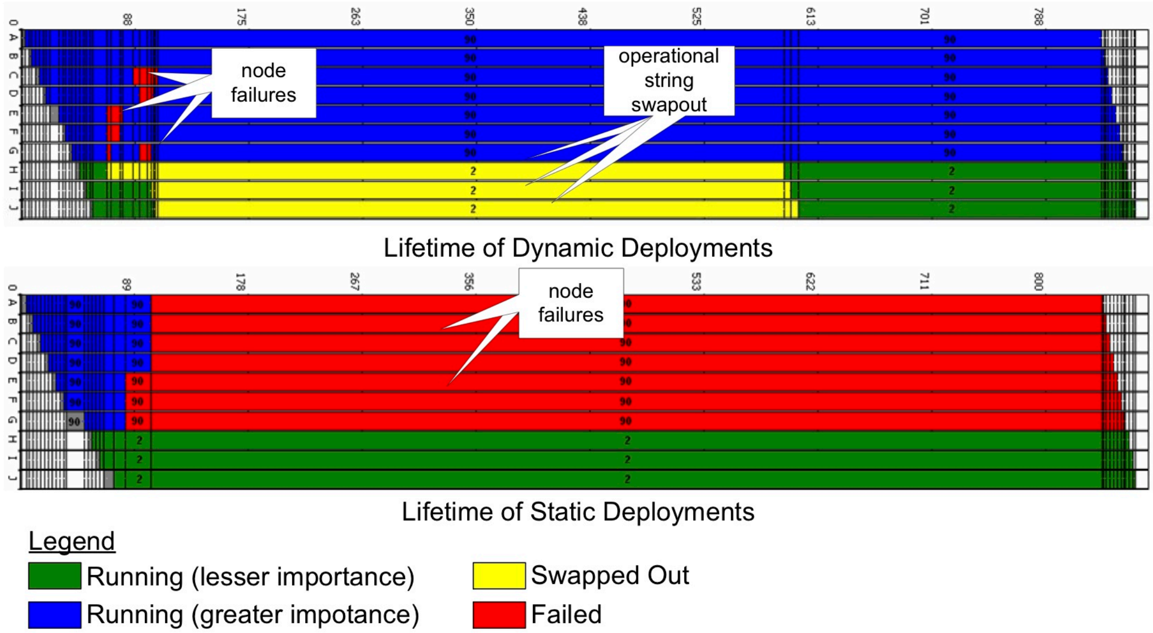

Figure 4. Graphical analysis of static Deployments (bottom) vs. dynamic deployments (top) using RACE.

The graphs in Figure 4, which are specific to RACE, were generated from the log messages stored in the database via the CUTS Logging Client and CUTS Logging Server described in the previous section. The x-axis in both graphs is the timeline for the test in seconds and each horizontal bar represents the lifetime of an operational string, *i.e.*, operational string A-J.

The graph at the bottom of Figure 7 depicts RACE's behavior when deploying and managing human-generated static deployment of operational string A-J. The graph at the top of Figure 4 depicts RACE's behavior when deploying and managing RACE-generated dynamic deployment of operational string A-J. At approximately 100 and 130 seconds into the test run, the CUTS Test Manager killed 2 nodes hosting the higher importance operational strings, which is highlighted by the “node failures” callout. This was accomplished by sending a kill command to the corresponding CUTS Node Manager.

As shown in the static deployment (bottom graph) of Figure 4, static deployments are not aware of the environmental changes. All operational strings on failed nodes (*i.e.*, operational string A-G) therefore remain in the failed state until they are manually redeployed. In this test run, however, we did not redeploy the operational strings hosted on the failed nodes because the random “think time” required to manually create a deployment and configuration for the 7 failed operational strings exceeded the duration of the test. This result signified that in some cases it is too hard to derive new deployments due to stringent resource requirements and scarce resource availability.

The behavior of dynamic deployment (top graph) is different than the static deployment (bottom graph) behavior. In particular, when the CUTS Test Manager kills the same nodes at approximately the same time (*i.e.*, section highlighted by the "node failure" callout), RACE's monitoring agents detect the environmental changes. RACE then quickly tears down the lower importance operational strings (*i.e.*, the section highlighted by the "operational string swapout") and redeploys the higher importance operational strings in their place (*e.g.*, the regions after the "node failure" regions).

The test run shown in Figure 4, however, does not recover the failed nodes to emulate the condition where the nodes cannot be recovered (*e.g.*, due to faulty hardware). This failure prevented RACE from redeploying the lower importance operational strings because there were not enough resources available. Moreover, RACE must ensure the lifetime of the higher importance operational strings is greater than lower importance operational strings. If the failed nodes were recovered, however, RACE would attempt to redeploy the lower importance operational strings. Figure 4 also shows the lifetime of higher importance operational strings was ~15% greater than lower importance operational string. This test case showed that RACE could improve the lifetime of operational strings deployed and managed dynamically vs. statically.

The results described above validate H1, *i.e.*, that CiCUTS enables developer to understand the behavior and performance of infrastructure-level applications in Emulab. Without CiCUTS, we would have used *ad hoc* techniques, such as manually inspecting execution trace logs distributed across multiple hosts in the test cloud, to determine the exact behavior of RACE. By using CiCUTS, however, we collected the necessary log messages in a central location and used them to determine the exact behavior of RACE.

If we did not use CiCUTS to validate RACE within Emulab, we would have had to manually execute the different actions required to emulate different scenarios, such as deploying each operational strings and killing/\-recovering nodes. Moreover, each action in the scenario requires precise timing of execution, which would have been difficult and inefficient to do manually. Because CiCUTS uses an automated approach to testing via the CUTS Test Manager, it is clear that CiCUTS can simplify validating the QoS of large-scale DRE systems within a test cloud, such as Emulab. This result, however, is for a single experiment run. To fully determine if CiCUTS can simplify validating the QoS of large-scale DRE systems within test clouds, we need to run large numbers of tests, *i.e.*, validate H2.

Using CiCUTS to ensure performance is within QoS specification. H2 conjectured that CiCUTS would help developers use test clouds to ensure the QoS of infrastructure-level applications is within its performance specifications throughout the development lifecycle. The results described above, however, represent a single test run of the baseline experiment. Although this result is promising, it does not show conclusively that CiCUTS can ensure RACE is within its QoS specifications as we develop and release revisions of RACE. We therefore integrated CiCUTS with the CruiseControl.NET (ccnet.thoughtworks.com), which is a continuous integration system (Fowler) we used to continuously execute variations of the experiment previously discussed while we evolved RACE. Figure 5 highlights the maximum number of tests we captured from the baseline scenario after it was executed approximately 427 times over a 2-week period.

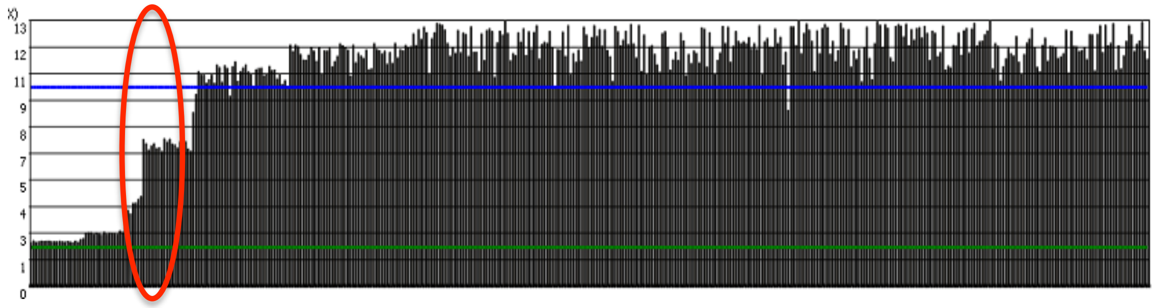

Figure 5. Overview analysis of continuously executing the RACE baseline scenario.

The number of executions corresponds to the number of times a modification (such as a bug fix or an added feature to RACE) was detected in the source code repository at 30-minute intervals. The vertical bars in Figure 5 represent the factor of improvement of dynamic deployments vs. static deployments. The lower horizontal line was the acceptable measured improvement and the upper horizontal line was the target improvement (i.e., 10%).

The heights of the bars in this figure are low on the left side and high on the right side, which stem from the fact that the initial development stages of RACE had limited capability to handle dynamic (re-)configuration of operational strings. As RACE's implementation improved – and the modified code was committed to the RACE source code repository – CruiseControl.NET updated the testing environment with the latest version of RACE, and then executed the CUTS Test Manager. The CUTS Test Manager then executed a test scenario in the Emulab test cloud. Finally, the circle portion in Figure 5 is where we located an error in the specification because the measured improvements were not correlating correctly with what we were seeing in the graphs produced by the system execution traces (see Figure 4). At this point, we learned that the equation for measuring RACE's improvement was incorrect due to a misunderstanding of the system's behavior and semantics in the target environment. After correcting the equation, with permission from the managers, we were able to meet the 10% target improvement.

The results in Figure 5 show how CiCUTS allowed developers to run test in Emulab and keep track of RACE's performance throughout its development. As the performance of RACE improved between source code modifications, the vertical bars increased in height. Likewise, as the performance of RACE decreased between source code modifications, the vertical bars decreased in height. Lastly, since each vertical bar corresponds to a single test run, if the performance of RACE changed between tests runs, developers could look at the graphical display for a single test run (see Figure 4) to further investigate RACE's behavior. These results therefore validate H2, *i.e.*, that CiCUTS helps developers ensure the QoS of infrastructure-level applications is within its performance specifications throughout the development lifecycle. As modifications where checked into the source code repository, the CruiseControl.NET detected the modifications and reran the CiCUTS tests for RACE.

CONCLUDING REMARKS

This chapter presented an approach for using test clouds to evaluate QoS properties of DRE system during early phases of the software lifecycle. Although it is possible to use test clouds for early QoS testing, test clouds alone do not always provide the necessary infrastructure to support such tasks. As shown throughout this chapter, we had to extend the CUTS SEM tool with logging and test management infrastructure to evaluate the RACE baseline scenario in the Emulab test cloud.

Although we put in time and effort to implement the necessary infrastructure to support our testing needs in the Emulab test cloud, the payoff was worth it because (1) we did not have to purchase the resources to meet our testing needs and (2) we were able to locate errors in RACE's specification and resolve them during its early stages of development. These results do not prove definitively that test clouds are the solution to all testing problems, similar to what we experienced with RACE. The results do show, however, that test clouds have the potential to address key needs in the testing domain if the correct infrastructure is available to support the testing requirements.

The CUTS SEM tool presented in this chapter is freely available in open-source format for download from URL http://cuts.cs.iupui.edu.

Bibliography

Deng, Gan, et al. "QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems." Son, I. Lee and J. Leung and S. Handbook of Real-Time and Embedded Systems. CRC Press, 2007.

Fowler, Robert. Continuous Integration. May 2006. <http://www.martinfowler.com/articles/ continuousIntegration.html>.

Gokhale, Aniruddha, et al. "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications." Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA) 73.1 (2008): 39-58.

Hill, James H. and Aniruddha Gokhale. Using Generative Programming to Enhance Reuse in Visitor Pattern-based DSML Model Interpreters. Institute for Software Integrated Systems. Nashville: Vanderbilt University, 2007.

Hill, James H., et al. "Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS." 12th International Conference on Embedded and Real-Time Computing Systems andApplications. IEEE, 2006.

Hill, James H., Sumant Tambe and Aniruddha Gokhale. "Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems." 14th

International Conference and Workshop on the Engineering of Computer Based Systems. Tucson: IEEE, 2007. 307-316.

Hill, James, et al. "Tools for Continuously Evaluating Distributed System Qualities." IEEE Software 27.4 (2010): 65-71.

Institute, Software Engineering. Ultra-Large-Scale Systems: Software Challenge of the Future. Tech Report. Carnegie Mellon University. Pittsburgh: Carnegie Mellon University, 2006.

Ledeczi, Akos, et al. "Composing Domain-Specific Design Environments." Computer 34.11 (2001): 44-51.

"Lightweight CORBA Component Model RFP." 2002.

Mann, Joan. The Role of Project Escalation in Explaining Runaway Information Systems Development Projects: A Field Study. Georgia State University. Atlanta, GA, 1996.

Porter, A., et al. "Skoll: A Process and Infrastructure for Continuous Quality Assurance." IEEE Transactions on Software Engineering (2007): 510-525.

Ricci, Robert, Chris Alfred and Jay Lepreau. "A Solver for the Network Testbed Mapping Problem." SIGCOMM Computer Communications Review 33.2 (2003): 30-44.

Rittel, H. and M. Webber. "Dilemmas in a General Theory of Planning." Policy Sciences (1973): 155-169.

Schmidt, Douglas C. "ACE: an Object-Oriented Framework for Developing Distributed Applications." Proceedings of the 6th USENIX C++ Technical Conference. USENIX Association, 1994.

Shankaran, Nishanth, et al. "The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems." 10th International Symposium on Object/Component/Service-oriented Real-timeDistributed Computing. Santorini Island, Greece: IEEE, 2007.

Smith, Connie and Lloyd Williams. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Boston: Addison-Wesley Professional, n.d.

Vouk, Mladen A. "Cloud computing — Issues, Research and Implementations." 30th International Conference on Information Technology Interfaces. Cavtat, Croatia, 2008. 31-40 .

Yu, Lian, et al. "A Framework of Testing as a Service." Information System Management. 2009.

—. "Testing as a Service over Cloud ." International Symposium on Service Oriented System Engineering. Nanjing : IEEE, 2010. 181 - 188 .