Versteeg, S., Hine, C., Schneider, J.-G., & Han, J. (2012). Emulation of cloud-scale environments for scalability testing.


Available from: [http://dx.doi.org/10.1109/QSIC.2012.57](http://dx.doi.org/10.1109/QSIC.2012.57).

Copyright © 2012 IEEE.

This is the author's version of the work, posted here with the permission of the publisher for your personal use. No further distribution is permitted. You may also be able to access the published version from your library. The definitive version is available at [http://ieeexplore.ieee.org/](http://ieeexplore.ieee.org/).
Emulation of Cloud-Scale Environments for Scalability Testing

Steve Versteeg,* Cameron Hine,† Jean-Guy Schneider† and Jun Han†

*CA Labs
CA Technologies, Melbourne, Australia
Email: steve.versteeg@ca.com
†Faculty of ICT
Swinburne University of Technology, Australia
Email: {chine, jschneider, jhan}@swin.edu.au

Abstract—Cloud computing increases the level of connectivity between software applications. IT management applications delivered as a service may need to connect to tens of thousands of endpoint systems. In order to validate the application’s reliability and performance at these very large scales, its scalability needs to be tested before being deployed in the cloud. We use an emulation approach, whereby endpoints are modelled and then executed in an emulation environment, which we call “Kaluta”. The key aspect is to balance the modelling of the endpoint systems such that it is rich enough to “fool” an unmodified application-under-test into thinking that it is talking to real systems, but lightweight enough such that tens of thousands of instances of model systems can be executed simultaneously in the emulation engine. We present an industry case study – CA IdentityMinder™-as-a-Service – to demonstrate the effectiveness of using emulation to validate the scalability of a cloud hosted application.

I. INTRODUCTION

The behaviour of a software system, is governed not only by its own implementation and internal state but also by the interactions it makes with users, devices and other systems in its environment. The trend is for IT systems to be increasingly interconnected. In a large enterprise, an IT management application, such as an identity management system, may need to connect to thousands or tens of thousands of other systems running on different servers disparately located around the world. With the advent of cloud computing, the scale of interconnectiveness has increased further still.

For the purposes of software quality assurance, it is necessary to validate that an application will meet the scalability requirements of its production environment. Scalability information is also useful to IT service architects for planning IT implementations. Knowing the resource usage which a software component will consume at a given scale allows service architects to accurately provision the appropriate hardware and deploy the right number of instances of a software component.

However, measuring the performance of a software component running in a very large scale in a test environment is a challenging task. A server application will not only have clients connecting to it, but may also need to make calls to other server applications (as shown in figure 1.) Situations where the component under test needs to connect to thousands or tens of thousands of other servers (also called endpoints) are particularly hard to test. While various load generation tools exist, these are generally designed to generate a large client load on a server application, but situations where a server application needs to make calls to thousands of other server applications, are generally not covered. Test environments with tens of thousands of physical servers are prohibitively expensive. The typical approaches used to measure scalability in test environments are to use virtual machines running on hypervisors such as VMware[1], [2] or Xen[3], or to use custom coded drivers and stubs. However, provisioning thousands of virtual machines is very resource intensive. (An overview of various approaches is given in Section II.)

In this paper we utilise an alternative approach. We demonstrate how a reactive enterprise environment emulator, we call Kaluta, can be used to emulate a cloud-scale environment with which an IT management software system can interact. We use Kaluta to measure the scalability of a real cloud-hosted enterprise management software application, CA Identity-Minder-as-a-Service[4].

II.RELATED WORK

A common approach to performance testing is to use load generation tools, such as HP LoadRunner[5] and the
SLAMD Distributed Load Generation Engine[6]. These tools are primarily designed for creating a large number of mock clients (also called ‘virtual users’) to generate a load on a server.

Mocking based approaches[7], [8], such as drivers and stubs, and associated tools are often used for unit testing, but are not suitable for system-level scalability evaluation as they abstract away from complexities such as communication infrastructure which may significantly impact on results encountered in deployments. Furthermore these approaches require large amounts of custom programming, making them difficult to generalise.

Virtual machines, running on hypervisors such as VMware Workstation[1] or ESXi[2], Xen[3] or VirtualBox[9], are a common approach used by Quality Assurance teams to test an application’s interaction with endpoint systems. Virtual machines create an entire virtualised hardware, software and application stack. This makes it very resource intensive to provision tens of thousands of virtual machines for the purpose of large scale testing. While it varies depending on workloads, the general rule of thumb is a virtual CPU to physical core ratio in the order of ten to one is the practical upper limit.[10] There are also maximum limits imposed by the hypervisor software itself. For example in VMware ESXi 5.0, the number of virtual machines per physical core has a maximum limit of 25, and the total number of virtual machines per physical host is limited to 512.[11]

A lighter weight approach to virtualisation is at the operating system-level, such as Solaris Zones[12] and Linux Containers (LXC)[13]. These virtualisation technologies allow applications to run on the same instance of an operating system, but in isolated virtual containers. This lets the hardware and operating system stacks be shared amongst containers, thereby using less resources than full virtual machines. Currently, operating system-level virtualisation is only available in a subset of operating systems.

The words ‘emulation’ and ‘simulation’ are often used interchangeably. In the context of this paper, we mean emulation to mean the run-time replacement of a component or components with executable models which imitate the behaviour of the real components. We use simulation to refer to the modelling of a component’s characteristics in order to make predictions about its behaviour. The clear distinction is that these models are not executed to replace the real components.

There are many simulators and emulators available for computer network analysis and testing. One example is ns-3[14], a widely used framework for network simulation and emulation. ns-3 supports the simulation of complex network topologies and large scale statistical experiments. Network simulation tools are aimed at the packet level, and are generally not concerned with the contents of packet payloads. This means that their direct use in application layer emulation is limited.

There are a number of initiatives to create modelling and simulation tools for performance engineering. For example, ePasa[15] is a performance modelling and simulation tool targeted at service oriented architectures. By creating detailed performance models of the various components in a large system, ePasa predicts where key performance bottlenecks may occur. [16] propose modelling techniques for simulating the performance of large systems. Their models use historical performance data (such as CPU time) about various components and use Kalman filters to forecast emerging performance issues. We see emulators and simulators as being complementary. Emulators can be used to provide additional quantitative data with which to calibrate simulation models.

Kaluta’s approach is most closely related to other application-layer emulation efforts. ITKO LISA[17] is a commercial tool; its Service Virtualisation automatically creates models from the observed message traces between a component-under-test and the other services in the environment. This enables organisations to accurately test the impact of new versions of a system without affecting the production environment and without needing to access the real systems in the production environment. Another emulator is the SoftArch/MTE[18] system, which is able to generate executable distributed test-beds from high-level models of the system. The SoftArch/MTE test-bed is intended help evaluate the performance of different architectures, design and deployment choices for a system while it is being developed. The focus of Kaluta is enabling the measurement of a software component’s scalability, with respect to it connecting to a very large number of endpoint systems, by emulating these endpoints on a single physical host.

III. CASE STUDY

The CA IdentityMinder™ (IM) software system [19] is an enterprise software application which manages users, roles and identities and their access to resources in large organisations. IM provides a logically central management point for controlling access to resources across many different types of systems which may exist in an organisation. Example types of systems may include: mainframes, databases, LDAP directories, operating systems, enterprise resource planning systems and cloud services. The systems administrated by IM are called endpoint systems.

People in the organisation have identities which may be associated with multiple user-names on different endpoint systems. For example John Smith may have a username of johns for his database account while his Windows domain login might be jsmith. IM allows these multiple accounts to be mapped to the same identity. The administration model of IM supports role based access control (RBAC) [20]. This simplifies administration by having users assigned to roles, which in turn are associated with resources on endpoint systems.

The IdentityMinder system supports CRUD (create, read, update and delete) operations on a variety of objects on endpoint systems. Some example basic operations include:

- Acquiring an endpoint, this allows IM to manage the endpoint system;
- Creating a new user account;
• Updating a user account, for example, changing a password; and
• Removing or suspending a user account.

The IdentityMinder software system has a component based architecture. The Identity Manager Server (IMS) is a Java EE application which supports the higher level operations of role and identity management. It also provides a web based graphical user interface for managing roles and identities. The operations for provisioning and updating user account objects are handled by the provisioning server. Connectors translate provisioning operations to system specific commands on the endpoint systems. The provisioning server does not communicate to the endpoints directly but through a connector server, called the CA IAM Connector Server (CS). The CS provides a unified communication interface for a wide set of operations on a disparate set of endpoint systems. Each connector is a separate OSGi [21] bundle and may run within the same Java Virtual Machine (JVM) as the CS itself. Large scale implementations may deploy multiple instances of the CS for load balancing and fail-over. Figure 2 depicts a simplified component architecture for the IdentityMinder system.

![Simplified component architecture for the IdentityMinder system.](image)

The cloud based version of IdentityMinder is called IdentityMinder-as-a-Service (IMaaS)[4], part of the CloudMinder™ suite. IMaaS is hosted on an external cloud and provides identity management to organisations without requiring them to install and maintain all of the software on-premise. Organisations in the post-cloud world may now have a mixture of IT systems. Some of their IT services will be cloud hosted whereas others will still be on-premise. Therefore IMaaS needs to manage identities on both on-premise endpoints and on cloud hosted services.

A high level view of the IMaaS architecture is shown in Figure 3. The IMS and provisioning server components are hosted in the cloud. These connect to the Cloud connector server (CloudCS), which is a cloud hosted instance of the CS. It is impractical for the CloudCS to communicate directly with all the on-premise endpoint systems, as this would require a separate network connection for each endpoint, leading to thousands or tens of thousands of connections, each of which would require a corporate firewall exception. Instead the connections from the CloudCS to the endpoints are multiplexed through a secure tunnel. An on-premise instance of the CS (called the Enterprise CS) receives the communication and distributes it to the on-premise endpoints. As with an on-premise implementation of the IdentityMinder system, an IMaaS implementation may include multiple instances of the CloudCS or the on-premise Enterprise CS for load balancing and redundancy.

The CS, as both the Cloud CS and the Enterprise CS, is the component which requires the greatest scalability as it is responsible for communicating with the endpoints. It may be required to manage the connections to thousands or even tens of thousands of endpoint systems.

It is necessary to verify that the CS meets these scalability requirements before being it is deployed in production. Knowing the resource consumption at large scales is also useful to IT implementers as they can use this information to plan for the appropriate hardware and required number of CS instances, depending on the scalability requirements of a particular site. QA teams have a limited number of instances of all the different types of endpoints with which the CS may need to connect. However measuring the performance and resource consumption of the CS when it is managing up to tens of thousands of endpoint instances is a challenging task. The common approaches used by QA teams to address this problem are to use virtual machines running on VMware or Xen hypervisors, or to have the CS connect to stubs. However these approaches are not ideal. It is generally not feasible to get more than 10 virtual cores running per physical core, so tens of thousands of virtual machines still requires thousands of physical hosts. Writing stubs requires custom coding and may not accurately reflect endpoint behaviour.

In this paper we explore an alternative approach. Endpoints are modelled and executed using the Kaluta framework. We emulate 10,000 endpoints running on a single physical host machine, with each emulated endpoint having its own IP address. We then measure the scalability of an unmodified CS while it is managing these emulated endpoints.
IV. KALUTA – AN EMULATION FRAMEWORK

Kaluta is an emulation framework for modelling endpoint behaviour and executing it. The modelling targets the interaction that occurs between a software component under test (CUT) and the other systems with which it communicates (endpoints). The aim is to make the models rich enough to “fool” the CUT that it is talking to real systems, but light-weight enough such that thousands of instances can be executed on a single physical host. This balance is achieved by modelling only the subset of an endpoint’s behaviour which is observed to be invoked through interactions with the CUT. Other behaviours and the systems which sit underneath and in the background are ignored from the modelling where possible. Kaluta provides a framework for easily and compactly modelling endpoint behaviour. The details can be found in [22], [23], [24], this section presents a summary.

The modelling stack is divided into three major levels: protocol modelling, behavioural modelling and data modelling.

The protocol model describes the types of messages (also called shapes) which are exchanged between the CUT and an endpoint and the order in which they may be sent. Kaluta provides a modelling syntax for concisely specifying application layer protocols. The syntax supports concurrency, allowing for multiple requests to be processed in parallel. A key feature is dynamic extension which is necessary for concise modelling of subsidiary concurrent operations. Figure 4 gives an example, for the Lightweight Directory Access Protocol (LDAP) protocol.

The behaviour model provides the logic for how to compose the contents of messages. It is responsible for inspecting and comprehending incoming messages and for putting together outgoing messages which will make sense for the CUT. The behaviour model is also responsible for updating a model endpoint’s internal state. Endpoints can be modelled with differing degrees of detail and depth, which we call the fidelity of the model. In Kaluta, the behaviour modelling layer is implemented with Haskell[25] functions.

The final layer of Kaluta’s endpoint modelling stack is the data model. The behaviour model uses the data model to populate messages with real data values and may write to the data model to update the state of an endpoint.

Endpoint specification models are executed in the emulator. The architecture of the Kaluta emulator is shown in Figure 5. The emulator is implemented in two main components: the network interface and the engine. The network interface is responsible for handling network communication with the CUT (or multiple CUTs). Each emulated endpoint is bound to a separate server port and IP address. The network interface
listens on these server ports and establishes new connections, each instance of a connection is called a conduit. The network interface is also responsible for decoding incoming messages and encoding outgoing messages into the correct protocol format. Incoming messages are passed onto the engine, and outgoing messages are sent to the CUT. The network interface is implemented in Java.

The engine component is responsible for executing endpoint model specifications. Incoming messages are processed by the model endpoint instances. Messages may be checked for conformance by the protocol model, and then passed to the behaviour model, which will compose a response dependent on the behaviour model specification and the internal state of the model endpoint. The behaviour model may also update the internal state of the endpoint by writing to the data store. The Kaluta engine is implemented in Haskell.

The network interface and the engine communicate with each other via Apache Thrift. Having the network interface and engine as separate components increases the runtime flexibility. These components may be deployed on different hosts. It is also possible for the network interface to be connected to multiple engines (or vice versa.)

The Kaluta emulator is configured via configuration files. These files specify the network interface and engine instances and their network addresses. Each endpoint to be emulated is listed in the configuration file, which includes a reference to a model specification and the network address on which the endpoint should be bound.

V. Experiment Design

The CA IAM Connector Server (CS) and Kaluta were installed on separate physical hosts, both of which were server class machines. The CS host (Machine A) had a quad-core Intel Xeon X5355 CPU and 12GB of RAM. The operating system used was Windows Server 2008 R2 64-bit. The Kaluta host (Machine B) was a Dell PE2950 with 2 quad-core Xeon X5355 processors and 24 GB of RAM. The operating system used was 64-bit Ubuntu Linux 11.10. The two machines were connected via a dedicated gigabit ethernet link over a Cisco packet switch.

We installed a development version of the CS. The build was taken from the forward development branch of IdentityMinder code-base which was being used to develop the IMaaS release. This was a pre general release version that was being subject to the regular quality assurance process. We installed this version of the CS on Machine A using default configuration settings. We then changed the start up parameters for the CS to increase the maximum heap size to 5GB to allow for a large number of endpoints to be managed. We used the CS’ JNDI connector to communicate with the Kaluta emulated LDAP endpoints.

A private subnet was set up between the Kaluta host machine and the CS host machine, and both machines were assigned static IP addresses. In order for the emulated endpoints to be managed separately they each needed a unique IP address. We achieved this by using the `ifconfig` GNU/Linux utility to acquire an additional 10,000 IP addresses on Machine B.

Kaluta was configured to emulate 10,000 LDAP endpoints. We used the high fidelity LDAP model described in Section IV. In the Kaluta configuration file we assigned each emulated endpoint a separate IP address.

The CS can be invoked through a LDAP API. We wrote an Apache JMeter[26] script to invoke the CS directly (taking the place of the IMS.) This allowed us to test the CS independently and made it easy to exercise the various aspects of the CS’
operations. The JMeter script sent LDAP requests to the CS, which orchestrated the CS to perform operations on the endpoint systems. The script initiated a series of actions which were representative of a typical set of tasks performed at the endpoints for identity management. These operations included:

- Send an LDAP bind request with administrator credentials to the CS. This allows subsequent JMeter requests to perform administrator level tasks.
- For each endpoint:
  - Acquire the endpoint by:
    - Sending an LDAP bind request to the endpoint with the administrator credentials of the endpoint system.
    - Exploring the endpoint to discover any manageable objects. During this process the CS sends a series of search requests to the endpoint.
  - Add a new user account to the endpoint.
  - Modify the password of the new user account.

Each operation results in the CS sending one or more LDAP requests to the endpoints under management.

The experiment was repeated using different numbers of concurrent JMeter threads during the execution. The number of concurrent threads ranged from a single thread to one hundred. Concurrent JMeter threads causes the CS to receive the JMeter LDAP requests concurrently and triggers the CS to communicate concurrently with the endpoint systems. When only a single JMeter thread is used, then all actions are processed sequentially, with the CS waiting for each action to be completed before initiating the next action, updating one endpoint at a time. Using multiple threads drives the CS to manage endpoints in parallel.

An overview of the experimental configuration is shown in Figure 6.

CA Application Performance Management (APM) version 9.1 was used to monitor the resource consumption of both Kaluta and the CS. We installed APM’s standard Java monitoring agent on the CS machine and a custom monitoring agent on the Kaluta machine to measure memory and resource consumption. Table I lists the resources measured.

### TABLE I

<table>
<thead>
<tr>
<th>Machine</th>
<th>Process</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>CS</td>
<td>CPU usage</td>
</tr>
<tr>
<td>A</td>
<td>CS</td>
<td>Java heap space used</td>
</tr>
<tr>
<td>A</td>
<td>CS</td>
<td>Total Java heap space allocated</td>
</tr>
<tr>
<td>B</td>
<td>Kaluta - Network Interface</td>
<td>CPU usage</td>
</tr>
<tr>
<td>B</td>
<td>Kaluta - Network Interface</td>
<td>Memory usage</td>
</tr>
<tr>
<td>B</td>
<td>Kaluta - Engine</td>
<td>CPU usage</td>
</tr>
<tr>
<td>B</td>
<td>Kaluta - Engine</td>
<td>Memory usage</td>
</tr>
</tbody>
</table>

#### VI. RESULTS

A. CS Results

The test described in Section V ran successfully. A single instance of the CA IAM Connector Server (CS) was able to successfully scale to manage 10,000 endpoints.

Table II shows the test completion times for a varying number of threads used in the JMeter script. For each number of threads, the experiments were run multiple times with completion times averaged. When using a single user thread, the CS will acquire and update the endpoints one at a time. Doing this, it took about 4 and a half hours to update 10,000 endpoints for the hardware and software configuration used in our experiment. Using two concurrent user threads almost halves the amount of time to complete the experiment. When 20 concurrent user threads were used, all of the endpoints were updated in about an hour and 20 minutes. This is an acceptable performance for acquiring and updating all the endpoints in a large organisation using only a single instance of the CS. Increasing the number of threads to 100 yielded no further speed-up in the completion time.

#### TABLE II

<table>
<thead>
<tr>
<th>JMeter threads</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>272.8 min</td>
</tr>
<tr>
<td>2</td>
<td>141.3 min</td>
</tr>
<tr>
<td>5</td>
<td>95.8 min</td>
</tr>
<tr>
<td>20</td>
<td>81.9 min</td>
</tr>
<tr>
<td>100</td>
<td>82.3 min</td>
</tr>
</tbody>
</table>

Figure 7 shows the rate at which the CS acquires and manages the endpoints for different numbers of user threads. For a single thread, endpoints are processed at a near constant rate over the course of the experiment, with approximately two endpoints acquired and updated every three seconds. For five JMeter threads, the endpoint acquisition rate is faster for the earlier acquired endpoints, before steadying out to a sustainable throughput rate, which is still faster than for the single threaded experiment. For 20 threads, the first 2000 endpoints are processed very quickly, taking just over four minutes, before the acquisition rate slows down to be about the same as for the five threaded experiment. There is no discernible difference between the acquisition rates of the 20 and 100 user thread experiments, implying that we have reached the throughput limit of the hardware and software.
configuration tested. With respect to the general trend of the rate of endpoint processing slowing down as the number of endpoints under management grows, we believe that this is at least partially due to the Kaluta emulator responding more slowly for larger numbers of active endpoints. In a separate experiment, we invoked Kaluta directly from the JMeter script, and observed a slowing down in response times when there are a large number endpoints which are active.

The rest of the results and figures reported refer to the 100 thread execution of the experiment.

As operating system network resources are finite, the CS will close the sockets of inactive connections. About ten minutes after the management operations are completed, the CS will automatically send an unbind request to the endpoint, as can be seen to be happening in Figure 8. However, even after the socket connection to an endpoint is released, the CS maintains a live reference to each endpoint under management, along with some metadata, to facilitate future communications with the endpoint. This information is maintained until the endpoint is explicitly removed from the list of endpoints under management (which was not done in this experiment.)

Figure 9a tracks the CPU utilisation of the CS along with system processes. The CPU usage pattern remains relatively consistent over the observed time period. The results indicate that the system is able to handle the workload of managing 10,000 endpoints without exhausting the available computation power. The CPU usage of the CS generally hovers at around 50%. When all processes (including JMeter) are considered, the total CPU usage exceeds 90% on only a few occasions.

The Java heap consumption of the CS is shown in Figure 9b. While the heap size fluctuates as resources are released and garbage collection takes place, it can be seen that the overall heap usage increases as more endpoints come under management.

Overall the results show that a single instance of the CS is able to manage 10,000 endpoint systems with acceptable performance.

B. Kaluta Results

Figure 9c shows the CPU usage of the Kaluta processes. The Kaluta network interface consumes the most CPU resources. There is a CPU-intensive burst of activity when the process starts, as it establishes the server sockets for the 10,000 IP addresses on which it listens on. As the CS acquires endpoints throughout the course of the experiment, the CPU usage of the Kaluta network interface is about 170-180%, or just less than 2 CPU cores. Most of this CPU activity is related to looping over the 10,000 server sockets to check for the establishment of new connections, and similarly, looping over the socket connections of the active endpoints to check for new incoming messages. The Kaluta engine CPU usage is minimal, consuming between 2-6% of a CPU core. Overall, the CPU usage of the Kaluta network interface and engine combined, is less than a quarter of the CPU power available on the host machine.
In terms of Kaluta’s memory use, the network interface and the engine both steadily increase their heap size as more endpoints become active over the course of the experiment. The network interface’s heap size reaches about 400MB by the end of the experiment, and the engine’s memory usage reached about 850MB. The total memory footprint of Kaluta, even after all 10,000 endpoints have been acquired, is about 1.3 gigabytes, which is relatively light compared to the 24GB of memory that was available on the server.

In total, for each test, 270,000 LDAP messages were exchanged between the CS and the emulated endpoints.

VII. DISCUSSION AND CONCLUSIONS

Kaluta demonstrates a new approach to measuring the scalability of a software component in an enterprise or cloud-scale environment. By emulating the interaction between a component and its environment, the scalability of a component in large scale environments can be tested.

We used Kaluta to conduct scalability testing of a real enterprise software component, the CA IAM Connector Server (CS), part of the cloud-hosted CA IdentityMinder-as-a-Service. Kaluta was used to emulate 10,000 endpoint systems for the CS to manage, representative of a large scale deployment. The emulation ran on a single physical server. The test confirmed that a single instance of the CS satisfactorily scales to manage at least 10,000 endpoints. We were also able to observe the CS’ resource consumption and performance characteristics at these large scales. This profiling information was passed onto the software architects, giving them information, which cannot be easily obtained in other types of testing. The software architects used this information to improve the design of the CS. For example, the CS used a third-party connection caching mechanism. This cache caused some memory to be consumed for connections even after they had become inactive and had been closed by the CS. Only by profiling at these large scales could this issue be found, as the memory usage of the cache is insignificant for a smaller number of managed endpoints.

Comparing to some other approaches, one common approach used by QA teams is to use virtual machines (such as VMware or Xen) as the test endpoints. The Kaluta emulation uses far fewer resources than emulating each endpoint with a virtual machine. The number of virtual machines which can run simultaneously on a physical machine is relatively small.
For example, VMware ESXi 5.0 has a maximum limit of 25 virtual machines per physical core.[11] Thus, emulating 10,000 endpoints in this manner would require at least 400 physical cores. Kaluta ran on a single physical server with 8 cores and used only a fraction of the computation resources available. In another test [23] we did a like-for-like comparison of Kaluta with a virtual machine approach and found that Kaluta uses 100 times less CPU computation time and 1000 times less memory.

We have shown that emulation is a feasible approach for large scale testing which fills a gap which is not easy to achieve with other tools. Software systems are becoming increasingly connected with other systems in their environment, and in a large enterprise this can involve connections to tens of thousands of other systems. It is important that software components are tested for the limits of their scalability before being deployed in production. With the increasing trend of cloud computing, software will be exposed to new scales. Giving developers the tools to measure performance at these extremes will be essential to delivering high quality cloud-enabled software.

A limitation of using Kaluta is that it is necessary to design and implement models for each type of endpoint system required. Future work will focus on automating this step by building models automatically from network traces. We are exploring using service virtualisation, as enabled by ITKO LISA, as a means of achieving this.

ACKNOWLEDGMENTS

The authors would like to thank Chen Chao and Janine Radford from CA Technologies’ quality assurance team for their extensive advice in conducting these experiments.

This work is supported by the ARC Linkage Project LP100100622 Large-Scale Emulation for Enterprise Software Systems.

REFERENCES