Software Service Emulation: Constructing Large-Scale Testing Environments

Cameron Hine, Jean-Guy Schneider, Jun Han
Swinburne University of Technology
Melbourne, Australia
Email: {chine,jschneider,jhan}@swin.edu.au

Steve Versteeg
CA Labs
Melbourne, Australia
Email: steve.versteeg@ca.com

***Abstract*—Constructing a testing environment for a software system that interacts with a large number (thousands) of other systems in production deployment is challenging, in particular in ascertaining its non-functional qualities such as scalability. Existing scalable testing environments mainly focus on the client perspective of the issue, i.e., generating scalable client workloads on the system under test. However, they largely ignore the service perspective concerning the large or scalable number of systems or services that the system under test uses during operation, which has substantially different characteristics. In this paper, we introduce *software service emulation* as an approach to address this shortcoming. More specifically, it consists of (i) a meta-modelling framework capable of modelling the services of any given system in the deployment environment of the system under test (including its messages, protocol, behaviour and states), and (ii) an emulation environment providing the run-time environment necessary to execute a large number of service models, mimicking the the behaviours and characteristics of the systems modeled for use by the system under test. A service emulation environment, KALUTA, is developed, and its scalability and ability to support large-scale testing activities is evaluated. We find that KALUTA is highly scalable, being able to concurrently emulate 10,000 LDAP directory servers using a single physical host. We has also succesfully used KALUTA in testing industry software systems, in particular facilitating the scalability testing of an enterprise-grade identity management suite: CA IdentityMinder.**

I. INTRODUCTION

Testing software systems which depend and rely upon interactions with other systems in a distributed environment is challenging. Testing environments which mimic the behaviour and characteristics of distributed environments are necessary for a system under test (SUT) to be tested in conditions resembling those encountered in production deployments. The sheer scale of some distributed environments makes it quite difficult to construct testing environments representative of production environments. In the enterprise domain, for instance, it is common for there to be tens-of-thousands of systems present within an environment. Constructing testing environments at such scales is inadequately supported by existing tools and approaches, but is crucial for system testing activities and to evaluate non-functional qualities of systems intended to operate in such environments.

Without such testing environments, the following questions remain largely unanswered:

- Is a SUT able to operate in environments of the scale expected in production?
- If a SUT can indeed operate at the requisite scale, how well does it do so, and at what scale does system performance begin to deteriorate significantly, if at all?
- If deployed into a specific production environment, will the SUT perform as advertised?

From a system under test's perspective, a deployment environment can be considered from two polar perspectives, as depicted in Figure 1: the *service*, or service *provider* perspective, and the *client*, or service *user/consumer* perspective. In the latter perspective, other systems/clients make requests to the SUT whereas in the former perspective, the SUT acts as the client and requests services from other systems. Ideally, testing environments allow a SUT to be tested from both perspectives, that is, as a service provider and as a service user, respectively.

Testing a SUT from a service *provider* perspective is well supported by existing tools. For example, load generating performance testing tools such as SLAMD, HP Load Runner, and Apache JMeter can generate scalable amounts of client load, imitating large numbers of concurrent users, and allow evaluation of a SUT's performance and scalability under those loads. However, none of these tools can operate in "reverse" mode and respond to incoming requests from a SUT.

Testing a SUT from a service *consumer* perspective requires an environment containing entities imitating the behaviour of software services which the SUT makes requests to. The mock objects approach [1] allows developers to define superficial implementations of external services. By implementing a mock object which meets a service's interface, a developer can code an approximation of the expected behaviour of the real service and use that mock object for testing purposes. This approach is popular with developers and is well supported by frameworks such as Mockito(Java), rspec(Ruby), and Mockery(PHP). However, by providing service imitations hooked directly into a SUT's code, calls to the underlying environment, such as the operating system's networking services, are typically bypassed and, consequently, are not appropriate for system testing purposes.

Another approach, widely adopted by industry, is the use of system-level virtual machines (VMs), such as those provided by VMWare, Virtual Box and Xen, as both, production and testing environments. Virtual machine environments are well

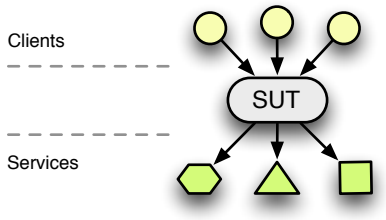

Fig. 1. Polar Perspectives of Software Testing Environments.

suited for system testing as their behaviour is essentially equivalent to their real counterparts, but are far easier to administer. However, scalability is a limiting factor. 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 as the practical upper limit [2]. This means that clusters of high-end machines are required to host environments containing tens-of-thousands of VMs, which is typically prohibitively expensive for testing purposes.

Service emulation is an approach we propose to constructing large scale testing environments which imitate interaction behaviour of software services. There are two key elements in the service emulation approach: (i) modelling approximations of real service behaviours to an adequate fidelity, that is, a level accurate enough to support the desired test scenarios, and (ii) an emulation environment capable of simultaneously executing such models, presenting the appearance and behaviour similar to real deployment environments. Properties of SUTs, such as scalability and performance, are evaluated whilst interacting with services provided by the emulation environment.

In this paper, we present three primary contributions resulting from our work on service emulation. (i) A layered service meta-modelling framework facilitating flexible service modelling. Flexibility is paramount so that the diverse needs of testing scenarios can be accommodated. An important aspect of this is the freedom to define environment behaviour up to the necessary level of fidelity; some tests require an environment with highly accurate behaviour, while for others, shallow approximations will suffice. (ii) An emulation environment supporting scalable service model execution and interaction with external SUTs. (iii) An empirical evaluation investigating the scalability and resource consumption of the emulation environment, a comparison with virtual machine approaches, as well as its effectiveness in industry testing scenarios.

The remainder of this paper is structured as follows: in Section III, we present our service meta-model and emulation environment KALUTA. Section IV describes our scalability experiments and use of KALUTA in testing a real enterprise software system. Section V discusses related work while Section VI concludes the paper, summarizing the major results and suggests areas for future work.

II. INDUSTRY SCENARIO

CA IdentityMinder (IM) [3] is an enterprise-grade identity management suite supporing management and provisioning of users, identities and roles in large organisations for a spectrum of different endpoint types. It is typically deployed into large corporations, such as banks and telecommunications providers, who use it to manage the digital identities of personnel as well as to control access of their vast and distributed computational resources and services. IM invokes services on the endpoints it manages in order to perform tasks such as:

- **Endpoint Acquisition:** Acquire endpoints (a computer resource) so that IM may then control its access and user permissions.
- **Endpoint Exploration:** Acquired endpoints require exploration to retrieve information required for IM activities. This essentially collates the manage-able identity and access objects on the endpoint so that IM can modify them as necessary.
- **Add Account:** User credentials are added to endpoints by IM.
- **Modifying Account:** Changes such as account passwords and permissions can be modified by IM and applied to affected endpoints.

IM is regularly deployed into environments containing tens-of-thousands of systems. Evaluating IM's run-time properties is important so that it can be confidently deployed into such large production environments. Performing these evaluations with every release may detect performance and/or scalability issues introduced by feature enhancements/additions or other code modifications. These evaluations may also be used by sales and marketing teams as a selling point for the product. Furthermore, IM deployment teams may use them as a way to guide production environment improvements: knowing which combination and configuration of resources can improve IM performance.

Evaluating IM's run-time properties requires test-bed environments containing tens-of-thousands of endpoints as results from evaluations in relatively small test-beds may not apply to large environments encountered in production. The behaviour of the test-bed must be accurate enough that IM can carry out its key operations (as listed above) without encountering significant behavioural anomalies. Finally, so that run-time issues caused by IM's use of networking and other operating system facilities may be detected, IM must be treated as opaque by the test-bed, that is, the test-bed cannot rely on hooks into IM's code to act as its deployment environment.

III. APPROACH

Service emulation provides large-scale environments, enabling non-functional testing of software systems prior to production deployment. The approach is based on the idea that *approximations* of service behaviour can be described using light-weight models executed by an emulation environment. SUTs can interact with emulated environments in the same manner they would with real deployment environments, allowing for its qualities, such as scalability and performance,

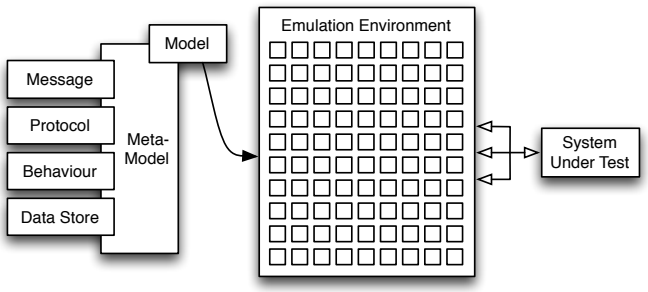

Fig. 2. Service Emulation Approach.

to be measured and evaluated. Key to service emulation is obtaining the right balance between behaviour approximation and scalability: the more complex the behaviour models become, the less likely are they to scale to the number of services typically found in large-scale distributed environments. We call this depth or level of behaviour approximation model *fidelity*. The higher the fidelity of a service model, the more accurate its behaviour.

Our service emulation approach is depicted in Figure 2 and is split into two key elements:

1) *Service Meta-Model and Models*: the service meta-model provides a framework for service models and allows their behaviour to be modelled to flexible levels of fidelity. This flexibility allows service modellers to meet the behaviour requirements necessary for the test scenarios in question. The service models themselves are defined as a composition of the elements provided by the meta-model and describe the behaviour of the emulated services.
2) *Emulation Environment*: the emulation environment executes a configurable number of service models (each corresponding to a real service) and, from the perspective of a SUT, appears to be a real environment. As show in Section IV, a single emulation environment can execute thousands of service models simultaneously using a single physical host and can thereby mimic large-scale environments (containing thousands of services or systems) using nominal computational resources.

SUTs interact with emulation environments in the same manner as real production environments, essentially unaware that they are interacting with an imitation, rather than real deployment environment. While operating, both the run-time properties (such as scalability and performance) but also the protocol conformance of the system under test can be measured and used to evaluate its non-functional qualities.

A. Service Meta-Model and Models

In order to model the behaviour of a real service, such as, for example, an LDAP directory server, a model capable of expressing such service behaviour is required. We define a service meta-model for this very purpose. The meta-model defines a structure and organisation for service models and allows modellers to define behaviour up to the level of fidelity most suitable for the testing scenarios in question.

There are four key elements in the service meta-model: *messages*, *protocols*, *behaviours* and *data stores*. Briefly, messages define *what* is communicated between systems, protocols define *when* certain messages can or cannot be transmitted, behaviours define *how* services respond to requests, and data stores define the persistent service level *state* which may be queried and modified by behaviour models when handling requests.

1) Messages: Distributed software systems interact with one another by exchanging messages transported using the underlying communication infrastructure. On-the-wire, messages may be encoded in a variety of different ways, encapsulated within HTTP chunks or by the basic encoding rules (BER) defined for ASN.1 specifications. So that emulated services may deal with messages of different encodings, we introduce a common message meta-model against which different encodings can be mapped.

In general, a message has two parts: a name or type, and a body or value. We refer to the name of a message as its *shape*. While different message values may have different structures, we generally model them as a *value* sequence. In particular, we tag or *label* values (called *associated* values) in a sequence in order to help identify and differentiate between values in sequences, while allowing other service specific treatment of values through a generic *base* value option. The following definitions define values and messages in our model.

Definition 1 (Values and Value Sequences)**.**

$$\begin{aligned} \mathcal{V} &\rightarrow l : \mathcal{V} && \textit{Associated Value} \\ &\mid \mathcal{B} && \textit{Base Value} \\ [\,\mathcal{V}\,] &= \{[\,]\} \cup \{[v_1, v_2, \ldots, v_n] : && \textit{Value Sequence} \\ &\qquad v_i \in \mathcal{V}, i \in \mathbb{N}\} \end{aligned}$$

where $\mathcal{B}$ denotes the set of base values, $\mathbb{N}$ is the set of positive integers, and l is an element of the set of labels $\mathcal{L}$.

Definition 2 (Message Meta-Model)**.**

$$\mathcal{M} = \varsigma \times [\,\mathcal{V}\,]$$

where $\mathcal{M}$ denotes the set of messages, ς the set of message shapes, and $[\,\mathcal{V}\,]$ the set of value sequences, respectively.

Example 1 below defines the *bind request* message in the *Lightweight Directory Access Protocol* (LDAP) [4]. LDAP is a typical *client-server* class protocol describing requests which a client may issue to a server and what responses it can expect, and is also the protocol we use in our case study system. The *bind request* message is sent by LDAP clients to authenticate at an LDAP server.

Example 1 (LDAP Bind Request)**.**

(bindRequest,
 [(messageID : 1), (protocolOp : (bindRequest :
  [(version : 3), (name : "dc=swin,dc=com"),
  (simple : ["user", "passwd"])]))])

The first element in this message is its shape, is bindRequest. The value sequence of the message itself begins with an associated value containing the lablel (messageID) and the value (1), which is used to identify the request (in particular, for use by the server in formulating the response message indicating the corresponding request message). The second value represents the protocol operation specified by the message. It is a series of nested associated values making up the bind request operation where the version is set to 3, the name is set to dc=swin,dc=com, the authentication is simple and has the value sequence ["user", "passwd"] as credentials. We use braces to group associated values so that the association boundaries are clear.

2) Protocols: In order for distributed software systems to interact with one another, they must adhere to some predetermined communication *protocol*. For the purpose of our work, we consider a protocol to define the *temporal order* of messages which are valid for exchange throughout system interactions. For an emulated service to approximate the behaviour of a real service, the communication protocol(s) used by this service must also be modelled. Without adhering to the expected communication protocol, emulated services will not interact in a protocol conformant manner and, therefore, will not accurately reflect the behaviour of real services. Adherence to the communication protocol ensures that, at the very least, an emulated service transmits messages of valid shape throughout interactions with SUTs. Furthermore, protocol models are also valuable in order to detect potential conformance issues in SUTs. In the following, we will outline the rationale of the protocol meta-model given in Definition 3.

The protocol meta-model is presented in Definition 3 in the form of an abstract syntax. Due to space limitations, we cannot discuss each element in detail and will rely on an example to compactly convey the flavour of the meta-model and its semantics. The interested reader can find detailed and more formal treatments of the model and semantics in [5].

Message *interactions* are the fundamental events defined for protocols. From a service's perspective, interactions cover both, the reception and transmission of a message, denoted by '?' and '!', respectively. For the sake of simplicity, our protocol meta-model defines message interactions using message shapes rather than message content. This allows protocol models to abstract away from message content details and focus on temporal concerns. Therefore, the protocol meta-model's interaction events are defined as direction/message shape pairs. All interaction events are annotated with a *continuation*, that is, a protocol model specification that defines what interactions, if any, are valid *after* the corresponding interaction event. Furthermore, the protocol meta-model enables non-determinism by incorporating *choice* between interaction events.

Definition 3 (Abstract Syntax for Protocol Meta-Model)**.**

$S \rightarrow D$	*Specification*
$\mid \mathcal{P}$	
$D \rightarrow v = \mathcal{P}$ **and** D	*Multi Declaration*
$\mid v = \mathcal{P}$ **in** $\mathcal{P}$	*Single Declaration*
$\mathcal{P} \rightarrow \mathcal{P} * \mathcal{P}$	*Product*
$\mid \mathcal{P} \sim \mathcal{P}$	*Extension*
$\mid I$	*Interaction*
$\mid v$	*Variable*
$\mid \mathbf{0}$	*Inaction*
$I \rightarrow I + I$	*Choice*
$\mid o\sigma.\mathcal{P}$	*Standard Interaction*
$\mid o\sigma \triangleright \mathcal{P}$	*Contractive Interaction*

where v is an element of V, the set of protocol variables, o denotes the direction of a message (transmission or reception), and σ is an element of the set of message shapes ς.

At a structural level, the protocol meta-model includes operations for standard and subservient parallelism. More specifically, the protocol meta-model allows for the specification of protocol *products*, that is, the parallel composition of (sub-)protocols. In such a composition, interaction sequences of two or more protocols are treated independantly. Furthermore, the protocol meta-model includes the notion of protocol *extension*. In contrast to protocol products, the extension(s) of a given protocol are not fully independant and may be terminated by interaction events occurring in the corresponding parent protocol, denoted as protocol *contraction*. These operations together enable concise specifications of subservient parallelism exhibited service protocols such as LDAP [4]. Finally, to facilitate the specifcation of more complex protocols, the meta-model includes the notion of protocol variables as well as (sub-)protocol declarations, respectively.

Example 2 demonstrates the service protocol meta-model by modelling the LDAP directory server protocol. To enhance readability, all occurrences of protocol variables are underlined. We specify the basic protocol functionality in $\underline{Base}$, the functionality of searching in $\underline{Search}$, and extend $\underline{Base}$ with $\underline{Search}$ whenever a new search request is received (LDAP allows for multiple searches to be executed concurrently). Similarly, in order to enable the non-blocking of an LDAP server in the context of processing administrative and data modifying requests, the $\underline{Base}$ protocol is extended with protocol specifications encoding the appropriate responses. Contractive interactions are used to terminate any pending operations when a BindRq or UnbindRq request is received.

Example 2 (LDAP Directory Service Protocol)**.**

$$
\begin{aligned}
\underline{Base} = \quad & ?\mathsf{UnbindRq} \triangleright \mathbf{0} \\
& + ?\mathsf{BindRq} \triangleright !\mathsf{BindRes}.\underline{Base} \\
& + ?\mathsf{SearchRq}.\underline{Base} \sim \underline{Search} \\
& + ?\mathsf{ModRq}.\underline{Base} \sim !\mathsf{ModRes}.\mathbf{0} \\
& + ?\mathsf{AddRq}.\underline{Base} \sim !\mathsf{AddRes}.\mathbf{0} \\
& + ?\mathsf{DelRq}.\underline{Base} \sim !\mathsf{DelRes}.\mathbf{0} \\
\textbf{and} & \\
\underline{Search} = \quad & !\mathsf{SearchEntry}.\underline{Search} \\
& + !\mathsf{SearchDone}.\mathbf{0} \\
\textbf{in } & \underline{Base}
\end{aligned}
$$

3) Behaviour: The main responsibility of services in distributed software environments is processing requests. Clients send requests to services which, in turn, process them and return a response (or a sequence of responses) conveying the result of the request. The purpose of the behaviour meta-model is to facilitate modelling of service behaviour.

There exist a multitude of ways how the behaviour of services can be modelled. The specific approach chosen largely depends on the level of *fidelity* required of an emulated service to support testing scenarios in question. Flexibility in the behaviour meta-model is crucial so that emulated services behave at the right level of accuracy required for the testing scenarios in question using the resources and service data available. This is achieved by allowing modellers to define request handler functions and bind these to the reception of specific message shapes through a dispatch dictionary.

The dispatch dictionary contains mappings between message shapes and request handler functions. Upon receiving a request with a certain shape, the corresponding handler function is retrieved from the dispatch dictionary and subsequently invoked, passing through the request as well as the current state of the service data store. The result of the handler function contains a sequence of messages, that is, the response sequence, which embodies the result of the request and, optionally, the updated state of the service data store.

Generic request handler functions can defined by service modellers, capable of handling requests of many different shapes. By binding generic handler functions to these different message shapes in the dispatch dictionary, the generic handlers can be reused and consequently reduce modelling effort.

The request handlers defined by service modellers need to satisfy the function signature specified by the handle-rq operation defined in Definition 4. An emulated service, upon receiving a message (request), invokes the corresponding handle-rq operation, passing along the request message for processing, paired with the current state of the service data store. Usually, the request processing results in a, possibly empty, sequence of responses. If the operation fails, however, error can be returned, although it is omitted here for simplicity.

Definition 4 (Request Handlers)**.**

$$\mathsf{handle\text{-}rq} : \mathcal{M} \times \mathcal{D} \to [\,\mathcal{M}\,] \times (\emptyset \cup \mathcal{D})$$

where $\mathcal{D}$ denotes the set of service data stores (cf. Section III-A4 below).

Table I presents an example dispatch dictionary for a modelled LDAP directory server. The BindRq and UnbindRq requests do not perform any complex operations. The result of an UnbindRq request is always empty sequence paired with empty set. The BindRq handler makes use of the generic function hdl-rq-skel, which creates response message based on response skeletons filled in by data provided by the corresponding request. Requests requiring non-trivial querying or modification of the service data store are another matter. The (AddRq, DelRq, ModRq, and SearchRq) requests are bound to custom handlers specifically coded.

Message Shape	Request Handler
AddRq	hdl-add-rq
BindRq	hdl-rq-skel
DelRq	hdl-del-rq
ModRq	hdl-mod-rq
SearchRq	hdl-search-rq
UnbindRq	([ ], ∅)

TABLE I
LDAP DISPATCH DICTIONARY.

Listing 1 presents an LDAP add request handler encoded in Haskell. There are three cases covered by this handler: (i) the entry is valid and may be added (lines 2-5), the (ii) entry already exists and cannot be added (lines 6-7), and (iii) there is no parent and the entry cannot be added (lines 8-9). In each case a message with the shape AddResS is returned with contents describing the result of the attempted add. In the case that all goes well, the updated directory data store is returned, otherwise an empty set indicates no state change.

```
1  handleAddRq ((AddRqS, [m_id, AddRq [dn, attrs ]]), ds)
2      -- valid entry to add
3    | not exists and p_exists =
4      ([( AddResS, [m_id, Success, parentOf dn ])],
5       insert dn attrs ds) -- insert into data store
6      -- entry already exists
7    | exists = ([( AddResS, [m_id, Exists, "" ])], empty)
8      -- parent does not exist
9    | not p_exists = ([( AddResS, [m_id, NoAttr, "" ])], empty)
10   where exists   = member dn ds
11         p_exists = member (parentOf dn) ds
```

Listing 1. LDAP Add Request Handler.

4) Data Store: Services in distributed software environments are often backed by a persistent data store. Web, email, file, database, and LDAP directory servers, for instance, are all backed by data stores. The behaviour of each of these services depends on the contents of these stores and the way in which that content changes over time. However, the structure and type of values within these data stores depends on the specific service: Web servers providing static web pages hold structured text of various kinds, relational databases

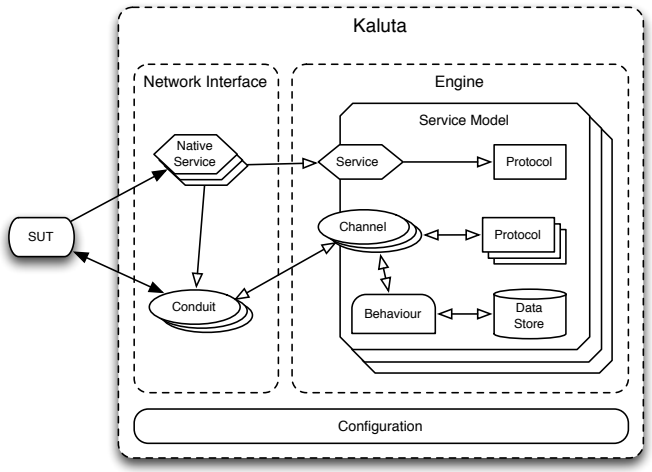

Fig. 3. KALUTA Architecture

hold tables of values of different kinds, and LDAP directories organise their data into hierarchical trees.

Therefore, it is crucial that the data store meta-model be flexible enough that these various structures and types can be expressed. This will make it possible for service behaviour models to better approximate the behaviour of real services. Additionally, however, it is not always desirable, or necessary, to have the exact same data store representations of a modelled service as would be used in a real service. Simpler data models can reduce modelling effort, allowing shortcuts in service behaviour models, trading accuracy for scalability and/or modelling time. In order to keep data store models as flexible as possible, and, therefore, able to support the range of different scenarios mentioned, we leave their definition open and simply use $\mathcal{D}$ to denote the set of data stores.

B. Emulation Environment

We have constructed a service emulation environment, which we call KALUTA, that, provided with a collection of service models, is able to present the appearance and behaviour of a distributed software environment. KALUTA's architecture is presented in Figure 3 and consists of three modules: a (i) network interface handling communication with SUTs, an (ii) engine which executes service models, and a (iii) configuration module to configure the network interface and engine modules.

1) Network Interface: The network interface allows communication with SUTs in a manner which is *native* to those SUTs. By native we mean that communication between the environment and SUTs occurs in the manner expected by those SUTs; messages are encoded on-the-wire according to the formatting requirements of the real service being emulated. The network interface acts as a bidirectional translator between the native messages transmitted over the communication infrastructure and messages understood by emulated services executed in the engine, respectively.

There are two key components of the network interface: (i) *native services* which allow SUTs to establish native communication channels for communication with the emulator, and (ii) *conduits* which associate network channels with engine channels as well as translate between the native message encodings required by the network (native) channels and the messages understood by the engine. Native services are bound to distinct IP address, port number pairs and listen for incoming connection requests. Upon receiving a connection requests from a SUT, the native service notifies the corresponding engine service passing along the relevant details. It also constructs a conduit to handle subsequent message based communications. Conduits are responsible for the exchange and transmission of messages on native channels and engine channels. Upon receiving a native message from an SUT, a conduit *decodes* it into the message structure understood by the engine and placed on the corresponding engine channel for processing. Similarly, when a message sequence response is received from the conduit's engine channel, the conduit *encodes* each message in the native on-the-wire format, transmitting them to the SUT via the corresponding native channel.

2) Engine: The role of the engine module is to execute multiple service models simultaneously. A service model has zero or more *channels* representing the communication mechanisms carrying messages between emulated services and external systems. Each service and channel is associated with a corresponding *protocol*. Channel protocols are maintained over the course of an emulation by the engine to reflect valid message receptions and transmissions. The behaviour and data store elements of the engine's service models correspond to the behaviour and data store layers of the meta-model. These are used by the engine to process valid requests.

Figure 4 presents the algorithm used in the engine to process message requests provided by the network interface which, ultimately, originated in an external software systems under test. The decoded message is represented by its pair of value sequence vs and message shape σ. The data store and dispatch dictionary of the service are denoted by d and dd, respectively. The current state of the channel's protocol is denoted by p while the constraint $\exists(\sigma, h) \in dd$ ensures that there is a handler in the dispatch dictionary which can process messages of the request's shape.

The first step of request processing is retrieving the dispatch handler. This is done by invoking lookup on the dispatch dictionary and passing in the request's message shape (Step 1). The request handler h is then invoked passing in the request (vs, σ) and the service's data store d, the result being stored in the pair (rs, d') where rs denotes the response sequence and d' the possibly updated state of the data store (Step 2). The message shapes of the response sequence are then used to retrieve the corresponding continuation in the protocol model by iteratively invoking next (Step 3). If the data store has been modified, that is, $d' \neq \emptyset$, then the service data store is updated (Step 4). Finally, the response sequence rs is returned to the network interface for encoding and transmission (Step 5).

Require: $(vs, \sigma) \in \mathcal{M}$, $d \in \mathcal{D}$, $p \in \mathcal{P}$,
$dd \in \varsigma \times$ handle-rq,
$\exists(\sigma, h) \in dd$

$h \leftarrow \mathsf{lookup}(\sigma, dd)$ ▷ Retrieve handler (1)
$(rs, d') \leftarrow h((vs, \sigma), d)$ ▷ Invoke handler (2)
for all σ' such that $(vs', \sigma') \in rs$ **do**
  $p \leftarrow \mathsf{next}(!\sigma', p)$ ▷ Update protocol (3)
end for
if $d' \neq \emptyset$ **then**
  $d \leftarrow d'$ ▷ Update the data store (4)
end if
  return rs ▷ Return response sequence (5)

Fig. 4. Request Processing

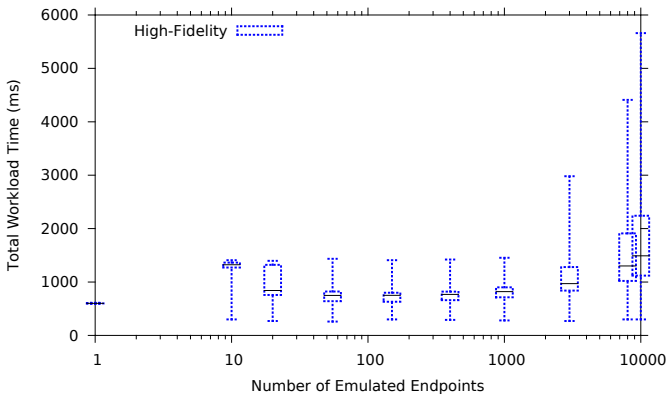

Fig. 5. Workload processing times for high fidelity KALUTA endpoints.

IV. EVALUATION

The scalability of the KALUTA emulator was evaluated with respect to three research questions:

RQ1: *Can* KALUTA *emulate emulate 10,000 endpoints on a single physical host and what is the resource consumption (CPU computation and memory usage) of the emulator for different emulation scales?*

RQ2: *How does the research consumption of* KALUTA *compare to one of the most common alternative approaches – that of using virtual machines (VMs)?*

RQ3: *What unique benefits can* KALUTA *bring to the testing of a SUT?*

A. Scalability of KALUTA (RQ1)

1) RQ1 Experimental Setup: A workload script was written to invoke a series of operations on LDAP directories. The operations conducted were typical of the types of operations an identity management system performs on the endpoints it manages. The sequence of operations were as follows: open network connection, bind to the LDAP directory, retrieve the whole directory through search, adding a new user, search a particular subtree of the directory, modify a user's password, search for a specific entry of the directory (verifying the preceding password modification), delete a user, and finally unbind. The workload script was executed with 32 concurrent user threads. Within each user thread, requests were sent synchronously, i.e. the response to the last request had to be received before the next request was sent.

KALUTA was installed on a Dell PE2950 server, with dual quad-core Intel Xeon E5440 2.83GHz CPUs and 24GB of RAM. The workload script was executed on another machine which had a dual core Intel Pentium 4 CPU and 2GB of RAM. Both machines ran the Ubuntu 11.04 64-bit as their operating systems. The two machines were connected via a 1 gigabit Ethernet connection.

We used a high fidelity model of an LDAP server. Each model directory server was backed by a datastore with 100 entries. We varied the number of LDAP directories emulated from a single endpoint to up to 10,000. Each emulated endpoint was given a separate IP address using the Linux `ifconfig` utility.

For each run of the experiment, we used CA Application Performance Management (APM) version 9.1 to monitor CPU consumption of the KALUTA processes, memory usage, and the total time taken to process the workloads. Each configuration of the experiment was run 6 times or more [6].

2) RQ1 Results: KALUTA successfully ran 10,000 emulated LDAP endpoints on the same physical host. Figure 5 shows the elapsed time taken to process each workload by the KALUTA emulator for increasing numbers of endpoints. For each emulated endpoint, about 125 LDAP messages were exchanged between the SUT and the endpoint. For 1000 endpoints or less, the median time taken to process a workload was 748 milliseconds. The total workload processing time becomes slower as the number of endpoints increases. For 2000 endpoints or greater, workload processing times start to increase and there is also a greater variability in the workload processing time. The median workload processing time per endpoint for 10,000 emulated endpoints is about 50% slower compared to the median processing times for smaller sized emulations of 1000 endpoints or less.

However, the response times of the KALUTA emulated endpoints, even for 10,000 emulated endpoints, was fast enough so as not to be the bottleneck in the testing of the IdentityMinder (IM) system as described in section IV-C. KALUTA was able to generate responses faster than IM was able to generate requests.

Figures 6 and 7 show the memory consumption and CPU usage of KALUTA, respectively. The peak memory consumption of KALUTA for 10,000 high fidelity endpoints was about 650MB. Note that this is significantly lower than the total amount of memory that was available on the host machine. The peak memory usage of the KALUTA engine increases linearly with the number of endpoints being emulated.

The total computation time for KALUTA emulations increases linearly with the number of emulated endpoints. Processing the workloads for the 10,000 endpoints consumed a total of 32 minutes of CPU time (spread across up to 8 cores of the host machine.)

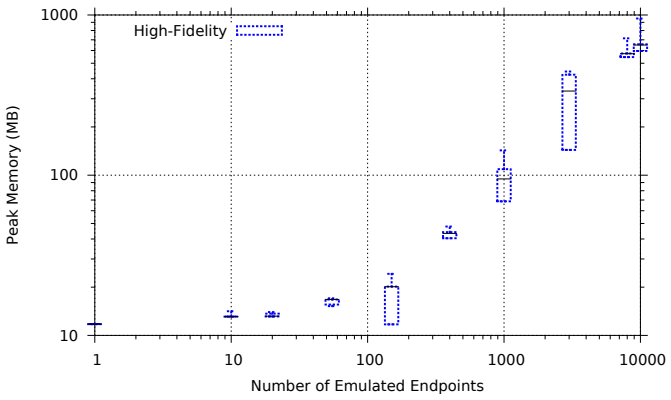

Fig. 6. KALUTA Peak Memory - Engine Module.

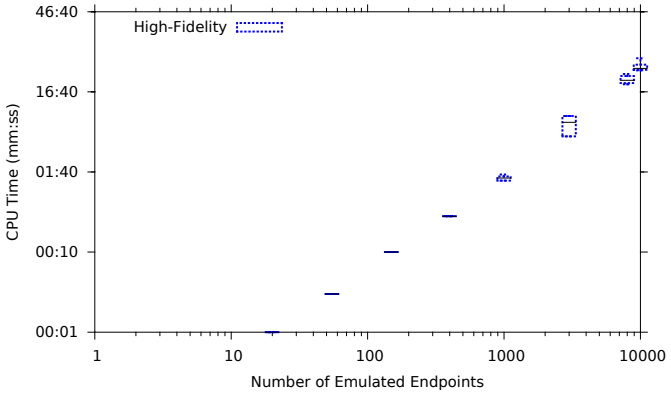

Fig. 7. KALUTA Total CPU Usage.

B. Comparison to VMs (RQ2)

1) RQ2 Experimental Setup: The VM experiments were conducted using the same workload script, hardware and environment as for RQ1. We used VMware Player version 4.0 as the hypervisor. We created a VM for each instance of an LDAP server endpoint. The VMs were run on the same host machine as KALUTA was installed on. For each VM we did a minimum installation of Ubuntu 11.04 64-bit and installed OpenLDAP Server. Each VM was allocated 128MB of main memory, which was the minimum we could reduce it to in order for the VM to boot. Each VM was given a 10GB virtual hard disk, which occupied about 1.7GB of physical disk space.

The recommended upper limit of virtual CPUs per physical core is between 8 and 10 [2]. Since the host machine had 8 physical cores, in order to ensure a high performance from the VMs, we stayed within the recommended limit and ran 60 VMs, each with 1 virtual CPU.

The workload was then applied to the 60 OpenLDAP Server applications running on separate VMs. We used APM to monitor the resource usage of the VM processes.

2) RQ2 Results: Table IV-B2 compares the performance and resource consumption of 60 VM endpoints to that of 60 KALUTA emulated endpoints. The resource usage of KALUTA is order of magnitudes less than that of the VM endpoints.

	VMs	KALUTA
Peak memory usage	11,006 MB	162 MB
Total CPU consumption	826.69 ms	9.29 ms
Hard disk space	102 GB	65 KB

TABLE II
COMPARISON OF RESOURCE USAGE OF VMS VERSUS KALUTA.

With respect to peak memory usage, KALUTA uses about 66 times less. In terms of CPU consumption, KALUTA uses about 90 times less. Finally, with respect to the hard disk space consumed, 60 VMs occupied over 100GB whereas the disk space taken by the KALUTA model specifications and configuration files was negligible.

C. Scalability Testing the CA IdentityMinder System (RQ3)

1) RQ3 Experimental Setup: We used KALUTA to measure the scalability of CA IdentityMinder (IM) (see Section II). The first requirement for KALUTA is that its models need to be accurate enough to 'fool' IM that it is interacting with real endpoints, which means that the responses returned need to be consistent with the responses IM expects from real endpoints. To validate this, we used IM's user interface to acquire a KALUTA emulated LDAP endpoint, explore it, add some users and modify some users. IM was able to perform these operations on the emulated endpoint without reporting any errors, which indicates that the emulated endpoints behaviour was consistent with IM's expectations of real endpoints.

We then setup an automated experiment to measure IM's scalability when managing up to 10,000 endpoints.

The CA IAM Connector Server (CS) is the component of IM which communicates with the endpoints and therefore requires the greatest scalability. We installed a development version of the CS on a separate machine to KALUTA. The CS machine had a quad-core Intel Xeon X5355 CPU and 12GB of RAM. The operating system used was Windows Server 2008 R2 64-bit. The only change made to the CS configuration from the installation defaults was to increase the maximum heap size to 5 GB. The CS machine and the KALUTA machine were connected via a 1 Gigabit Ethernet connection. KALUTA was configured to emulate 10,000 high fidelity LDAP endpoints.

We wrote a JMeter script to automate the Connector Server to invoke the same set of identity management operations on each endpoint as described in Section IV-A1. The JMeter script was run with different numbers of concurrent user threads, varied between using a single thread to up to 100. For each number of threads, the experiment was run at least six times with the results averaged. APM was again used to monitor the memory and CPU usage of the CS and KALUTA.

2) RQ3 Results: The test ran successfully and verified that a single instance of the CS can manage in excess of 10,000 endpoints. Table III lists the average completion times of the experiments for varying numbers of JMeter user threads. The single threaded experiment took 4 and a half hours to complete. Using 20 user threads reduced this to about 82 minutes. Adding more than 20 user threads did not further

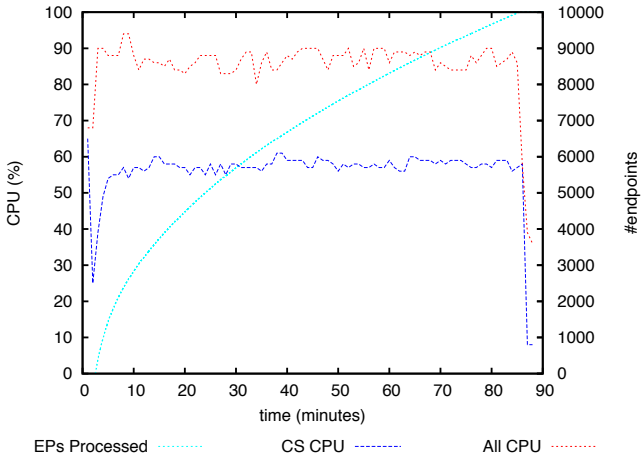

Fig. 8. Connector Server CPU usage.

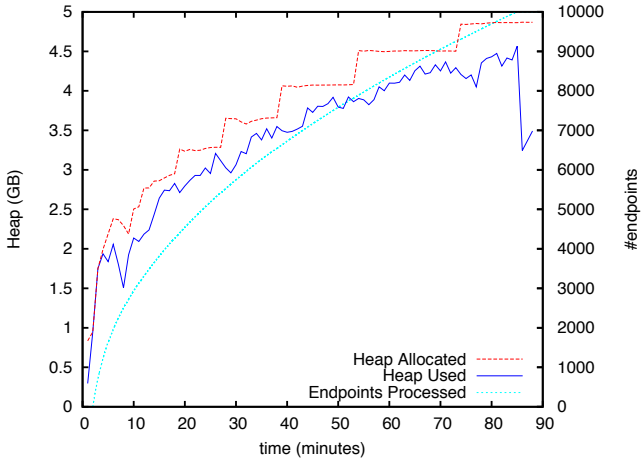

Fig. 9. Connector Server heap usage.

reduce the completion time of the experiment. These were acceptable completion times for acquiring and updating 10,000 endpoints, especially considering low end hardware was used.

JMeter threads	Total time
1	272.8 min
2	141.3 min
5	95.8 min
10	85.6 min
20	81.9 min
100	82.3 min

TABLE III
TIMES TO COMPLETE TESTS FOR 10,000 ENDPOINTS.

Figures 8 and 9 track the CPU utilisation and heap usage of the CS throughout the course of a sample run of the 20 user thread experiment. The CS uses 50-60% CPU over the course of the experiment. When all processes are included (which includes system processes and JMeter), total CPU usage sometimes gets up to 90%. With respect the heap usage of the CS, the heap size steadily grows as more endpoints come under management. When the full 10,000 endpoints have been acquired, the heap utilisation is about 4.5 GB.

KALUTA was able to handle all incoming requests generated by the CS, and its memory and CPU usage were consistent with the results reported in section IV-A2 for the 10,000 endpoint experiments. KALUTA's conformance checker confirmed that the CS sent no messages outside of the allowable protocol sequence. For each run of the experiment 270,000 LDAP messages were exchanged, with the conformance checker not once flagging a message which was out of sequence.

We also did some experiments to test how the CS handles protocol non-conformance from the endpoints. For example, we ran tests where KALUTA delayed its responses by up to 30 seconds or did not respond at all. The CS handled both the delayed responses and non-responses correctly.

D. Unique Insights into SUT

The large scale KALUTA emulations gave multiple insights with respect to the characteristics of IdentityMinder (IM) system operating at large scales.

Firstly, we were able to confirm that IM scales to manage at least 10,000 endpoints using only a single instance of the Connector Server (CS). Secondly, KALUTA allowed us to observe the resource consumption of the CS component of IM while operating at large scale. This profiling information was passed on to the software architects, giving them information which cannot be easily obtained through other forms of testing. The software architects used this information to improve the design of the CS. For example, one unexpected result was that the memory consumption of the CS was greater than expected. The underlying cause was found to be a third-party connection caching mechanism which was causing memory to be consumed for connections even after they had been closed. Only by running tests at large scales could this issue be found, since the memory consumption of the connection cache for a small number of endpoints is insignificant.

Thirdly, by collecting information about resource consumption and performance at large scales, we were able to provide information to IT implementers with respect to the system resources which will be required for a deployment of a given size. Fourthly, KALUTA automatically checked the CS for LDAP protocol conformance and observed no protocol violations for the millions of messages exchanged.

Finally, our testing showed that for large scale deployments, it is not only the software system itself which needs to be validated, but also the operating system and environment on which it is running. For example, when we deployed our workload generation script on Linux, we needed to increase the size of ARP (Address Resolution Protocol) cache table on the workload generation machine, in order for our test to complete in a timely fashion. This was because we were connecting to a greater number of different IP addresses within short succession than Linux was configured for by default.

E. Threats to Validity

There are potential threats to the validity of our work:

- The script which generated workload for KALUTA was executed with only 32 user threads due to hardware constraints. The maximum throughput of KALUTA was therefore not tested. For the SUT tested in this paper, the throughput of KALUTA was adequate, since the emulator could generate responses faster than the SUT generated requests. However, we cannot say whether KALUTA is sufficiently fast for faster SUTs. We plan to repeat our experiments using faster hardware for the workload.
- KALUTA endpoint models need to be manually specified for each different type of service protocol. This reduces the ease at which new types of services can be added to KALUTA. We are exploring using service virtualisation, as enabled by iTKO LISA, as a means of automating the model specification process.
- The experiments in this paper were on one type of service endpoint: LDAP. The heterogeneity of enterprise environments is one of the challenges of deploying software systems. We have validated the flexibility of our framework by modelling other protocols including Web Services and BitTorrent. Future work will include deploying multiple types of service models in parallel.

V. RELATED WORK

There are a number of works describing conceptual [7], [8] and formal [9] service models. Colombo, Di Nitto, Di Penta, Distante, and Zuccala [7] models services in terms of the core agents, actors and their relationships to one another. Our context, expressed the terminiology of thier model, consists of two primary agents, the SUT and the testing environment, playing the role (acting as) service consumer and service provider, respectively. KALUTA provides concrete simple services to the SUT which are lower fidelity than real services but suitable for testing purposes. In [8] Quartel, Steen, Pokraev and van Sinderen present COSMO: a conceptual service modelling framework supporting refinement. Our service model can be interpreted as focusing on a subset of the service aspects and abstraction level presented in COSMO. Namely a service model which focuses on the behavioural and information aspects of services at the choreography level of abstraction.

The role of KALUTA is similar to that of PUPPET [10]. PUPPET uses a model-based approach to generating stubs for Web Services. The functional behaviour of emulated Web Services are defined as Symbolic Transition Systems (STSs), or by UML 2.0 state machines which are translated into STSs for execution. The STS models on which functional behaviour is based encompasses the temporal, logic, and state aspects of service behaviour within a single model. Our service meta-model, on the other hand, segregates these three aspects into separate layers: the protocol, behaviour and data store layers. By separating these concerns we allow the possibility of different models to capture these properties to be mixed in at a later stage if beneficial. A protocol model based on Petri-netsor linear time logic may, for instance, be incorporated into later KALUTA versions. Another significant difference between PUPPET and our own work is our focus on scalability. Although PUPPET incorporates qualities through service level agreements into the testing environment, it does not set out to represent environments containing thousands of concurrent services.

VI. CONCLUSION

The scale of some large distributed environments makes it quite difficult to construct testing environments representative of production conditions. Service emulation is an approach we propose to constructing large scale testing environments. We have presented: (i) A layered service meta-modelling framework facilitating service modelling up to a flexible level of fidelity, accommodating the needs of different testing scenarios; (ii) KALUTA, an emulation environment supporting scalable service model execution and interaction with external SUTs; and (iii) An empirical evaluation investigating the scalability and resource consumption of KALUTA, comparing it with VM approaches, and investigating its effectiveness in industry testing scenarios. We find that KALUTA is substantially more scalable than virtual machine approaches, capable of emulating 10,000 LDAP directory servers using a single physical host, and effective in industry testing scenarios, providing unique insights into the run-time properties of CA IdentityMinder.

Future work includes automating aspects of service model construction alleviating dependence on human modelling. We also plan to connect a network emulator with KALUTA and investigate effects of different network configurations and topologies on SUTs.

REFERENCES

[1] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock Roles, not Objects," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2004, pp. 236–246.

[2] J. Sanchez. (2011) Squeezing virtual machines out [of] CPU cores. VM Install. [Online]. Available: http://www.vminstall.com/squeezing-virtual-machines-out-cpu-cores/

[3] M. Gardiner, "CA Identity Manager," November 2006, white Paper on CA Identity Manager.

[4] J. Sermersheim, "Lightweight Directory Access Protocol (LDAP): The Protocol," RFC 4511 (Proposed Standard), June 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4511.txt

[5] C. Hine, "Emulating Enterprise Software Environments," Ph.D. dissertation, Swinburne University of Technology, 2012.

[6] J. F. Box, "Guinness, Gosset, Fisher, and Small Samples," *Statistical Science*, vol. 2, no. 1, pp. 45–52, 1987.

[7] M. Colombo, E. Di Nitto, M. Di Penta, D. Distante, and M. Zuccal, "Speaking a common language: A conceptual model for describing service-oriented systems," in *Service-Oriented Computing - ICSOC 2005*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, and P. Traverso, Eds. Springer Berlin / Heidelberg, 2005, vol. 3826, pp. 48–60.

[8] D. Quartel, M. Steen, S. Pokraev, and M. van Sinderen, "Cosmo: A conceptual framework for service modelling and refinement," *Information Systems Frontiers*, vol. 9, pp. 225–244, 2007.

[9] M. Broy, I. H. Krüger, and M. Meisinger, "A formal model of services," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007. [Online]. Available: http://doi.acm.org/10.1145/1189748.1189753

[10] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini, "Model-Based Generation of Testbeds for Web Services," in *Testing of Software and Communicating Systems*, ser. Lecture Notes in Computer Science, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds. Springer Berlin / Heidelberg, 2008, vol. 5047, pp. 266–282.