

Scalable Emulation of Enterprise Systems

Cameron Hine, Jean-Guy Schneider, Jun Han
 Faculty of Information & Communication Technologies
 Swinburne University of Technology
 P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA
 {chine, jschneider, jhan}@swin.edu.au

Steve Versteeg
 CA Labs
 Level 2, 380 St. Kilda Rd
 Melbourne, VIC 3004, AUSTRALIA
 steven.versteeg@ca.com

Abstract

Testing enterprise software that communicates with a large number of other software systems is a challenging task as it is often difficult to replicate the size and heterogeneity of large enterprise environments. In particular, it is challenging to conduct testing of non-functional properties related to scalability, performance, and robustness of enterprise software systems when deployed into such environments. In order to address this issue, we propose the use of a scalable emulation environment enabling non-functional testing of enterprise software in realistic and large-scale settings. In this paper, we illustrate our approach by using deterministic finite state machines to specify a scalable and interactive emulation of a modeled environment. To demonstrate the practicality and scalability of the approach, a prototype tool is presented that is used to emulate a large-scale environment of up to 10,000 endpoint systems for an enterprise class software system.

1. Introduction

Modern enterprise software systems generally operate in complex environments of unprecedented scale and diversity and are often required to communicate with many thousands of distributed computer systems, each of which possessing its own unique configuration. A modern software system does not exist in isolation but is rather a component within a broad network of interacting systems. The behaviour of a software system is governed not only by the internals of the software itself but also driven by the interactions with other systems. With the increasing complexity of the environments enterprise systems are deployed in, it is getting more difficult to accurately represent realistic production environment communications during testing.

Nevertheless, businesses rely on such systems to manage critical aspects of their day-to-day activities, such as, for example, identity management and transaction processing. The combined factors of a complex operational environment and the often critical nature of the tasks being carried out makes it exceedingly important to rigorously test enterprise software systems before their deployment.

CA's *Identity Manager* [7], for example, is an enterprise software product used to oversee the identity and access management of software systems in large organizations. Organizations may have tens of thousands of computational resources located all over the globe. In order to perform its responsibilities, *Identity Manager* needs to regularly communicate with many, if not all, of the resources in its environment. Hence, it is necessary to test the possibly complex sequences of interactions with the resources prior to the deployment of systems such as *Identity Manager*.

A production environment is generally unsuitable to conduct this kind of testing as a fault in the enterprise software may cause disruption or irreversible damage to that production environment. Physically recreating the production environment is generally impractical for larger environments as there may be many thousands of systems, geographically distributed with varying individual configurations.

Testers therefore need to rely on limited recreations of the production environment and specialized tools which address specific facets of what the system may encounter in production. Virtual machines (such as those created with VMWare [16] or VirtualBox [18]) can provide a means to reuse particular system configurations. However, at present, they do not scale very efficiently due to significant resource requirements. Performance testing tools such as the SLAMD Distributed Load Generation Engine [12] or HP's LoadRunner [9] provide functionality which enables software testers to simulate many thousands of enterprise system clients using minimal resources. These tools however, have been designed for generating scalable *client* load against a server system under test. They are not generally concerned with generating server responses to client load, and server to server communication, which is precisely what is required to test enterprise software such as *Identity Manager*.

In order to address this shortcoming, we propose the use of a *reactive emulation environment* that replaces an enterprise system's deployment environment for non-functional testing. The main idea is to model the run-time behaviour of each system (henceforth known as *endpoint*) in the environment, based on the protocol specification(s) of the expected interaction(s) and replace each endpoint by an instantiation of the corresponding model in the emulation environment. In

order to achieve scalability, the endpoint models should be targeted towards mimicking *interaction behaviour*. As such, they should require as little computational expressive power as possible. The emulation environment itself should allow for an easy set-up and configuration of endpoint models and ideally run on one to a few real hardware machines only, but still be potent enough to enable the emulation of thousands of endpoints. We envisage that such an environment would allow software developers to conduct testing of enterprise software in a realistic deployment environment without needing to resort to physical creation of such an environment. It should be noted that the specification of test scenarios for enterprise systems is beyond the scope of this work; the proposed emulation environment will mainly provide an environment to *execute* such tests.

In this paper, we present some preliminary results obtained modeling a subset of the LDAP [15] directory server behaviour using a deterministic finite state machine. LDAP was selected for preliminary investigation for a number of reasons, including (i) the protocol specification is freely available, (ii) the protocol is somewhat complex, (iii) it is widely used in enterprise environments, and (iv) some types of requests may result in permanent (persistent) changes in response values.

The primary intention of this work is twofold: firstly, to uncover the main issues in modeling and emulating application layer interactions which occur in enterprise environments; secondly, to investigate the general feasibility of the emulation approach.

The remainder of this paper is structured as follows: in Section 2, we introduce the reactive emulation environment approach proposed in this work. Section 3 describes the LDAP protocol operations that we intend to model and emulate in the work. In Section 4, we discuss the relevant issues in regards to modeling of enterprise protocols and endpoint behaviour. Section 5 describes REACTO, a prototype implementation of a reactive emulation environment, which is followed in Section 6 by an illustration of using REACTO to emulate an environment for a connected enterprise software system. Section 7 describes the results of scalability testing conducted on the prototype system. Section 8 discusses related work and Section 9 summarizes our main observations and discusses future work.

2. Reactive Emulation in a Nutshell

In this work, we propose the concept of a scalable and reactive emulation environment as a means to enable testing of enterprise software in large-scale environments. The main goal of the environment is that it is *fully reactive* so that enterprise software may communicate with it in *real-time* without the need to modify the underlying enterprise software. In this section, we will further elaborate on the basic concepts of our approach.

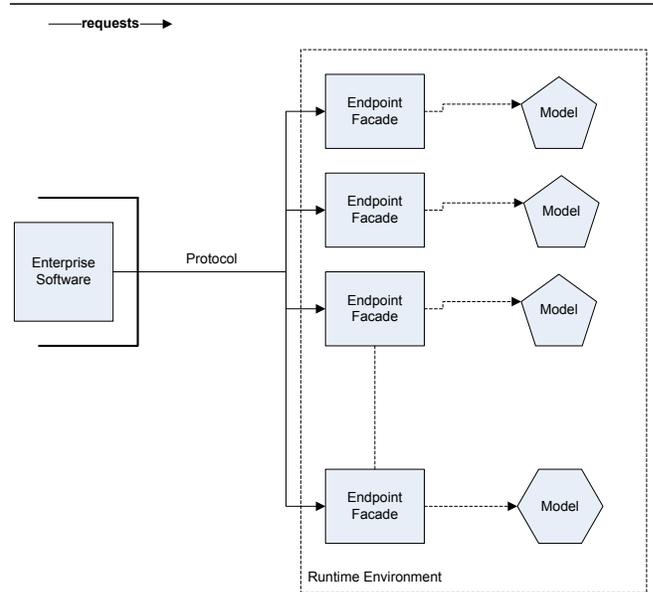


Figure 1. Conceptual Model of Emulation Approach.

Figure 1 illustrates the basic structure of the approach taken. The enterprise software system to be tested is depicted on the left-hand side, behind a bold solid line, indicating that the enterprise software is left unmodified, that is, it is unaware that it does not interact with endpoints of a “real” enterprise environment, but an emulation thereof. The arrows indicate the direction of *initial* communication; the enterprise software makes requests to the endpoints in its environment and not vice versa. In our approach, endpoints are represented as *endpoint facades*, meaning that from the perspective of the enterprise software system under test, an endpoint will appear as if it is in a real deployment environment. However, behind the facade there is no endpoint software being run, rather a corresponding *endpoint model* is used to dictate the behaviour of an emulated endpoint. The endpoint models may differ according to the kind of endpoint systems being modeled (represented in Figure 1 by the different shapes of endpoint models). It should be noted that an endpoint facade can not only receive requests intended for the associated endpoint system, but also send responses seeming to be sent from the intended endpoint.

The underlying runtime environment pulls the emulation together by providing the ability to execute endpoint models and allowing enterprise software to interact with the emulation environment. The runtime environment should also facilitate the configuration of the emulated deployment environment by providing a means to control the number of endpoint systems being emulated and what services are emulated on those systems.

Ideally, the reactive emulation environment shall be capable of emulating various protocols on a large scale, up to

10,000 emulated systems on a single (or very few) physical machine(s). This will have to be achieved by the use of endpoint models which capture the interaction behaviour of real enterprise environments, but require substantially less computational resources. The run-time behaviour of the environment will be primarily dictated by the models; testers of enterprise software will need to provide models for the types of endpoints they wish to emulate.

3. LDAP Emulation Requirements

LDAPv3 is an application layer protocol which supports 11 different server operations [15]. The operations fall into three different categories: *authentication*, *searching* and *modification*. In this work we mainly consider three operations, one operation from each category: (i) bind, (ii) search, and (iii) add. By choosing to investigate an operation from each category we ensure that issues arising out of each category are likely to be discovered. Also, these three operations are those required by the commercial tool we use in Section 6 to demonstrate the practicality of the approach.

The operation bind is from the authentication category. It is used by a client to connect to an LDAP directory using some authentication credentials. A bind request results in a bind response from the LDAP server. The bind response indicates whether the corresponding bind request was successful or not. After successfully binding to an LDAP server, search and modification operations for the client will be enabled.

The operation search allows a bound client to search an LDAP directory. The client issues a search request which contains a rich description of the search criteria. Searches may be restricted to certain levels of the directory tree, time and size limits may be specified on the directories responses, and search filters can be used to build powerful predicate based search criteria. Each search request issued by a client is matched by zero or more search result entries from the LDAP server and is finalized by a single search result done.

Finally, the operation add allows a bound client to insert an entry into an LDAP directory. An add request from a client is matched by a single add response from the server, indicating whether the corresponding request was successful or not.

4. Modeling

To realize the reactive emulation approach introduced in Section 2, we need to be able to address the following three issues: (i) modeling of protocols used by enterprise software systems to interact with endpoints (enterprise protocols), (ii) modeling of the enterprise endpoints, and (iii) creation of a run-time environment which can execute endpoint models and interact with enterprise software systems. In this section, we will describe our approach to address the first two issues,

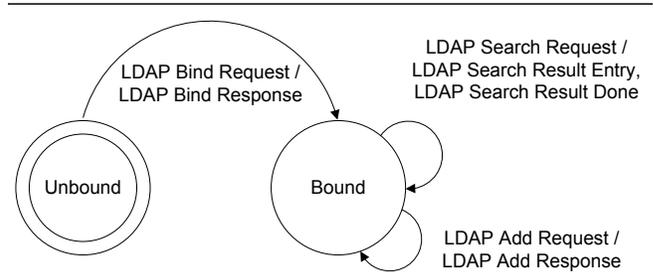


Figure 2. DFSM Model of LDAP.

Section 5 discusses our prototype implementation of the run-time environment.

4.1. Enterprise Protocols

Our preliminary model of enterprise protocols is based on a deterministic finite state machine (DFSM). Automata and their derivatives have been widely employed by researchers to model communication protocols [1], [2], [11]. These models have intuitive visual representations and are straight forward to translate into a fast and scalable executable model. DFSMs are fast because the executable model is basically a lookup table, and scalable because each emulated endpoint requires only a memory reference (pointer) to its current state in the table.

Our DFSM is a state machine which according to its current state and contents of its input stream may transition into another state, optionally producing a sequence of outputs. The input and output alphabets for the DFSM we propose are comprised of the set of unique input and output commands available for the protocol being modeled. These commands indicate the fundamental meaning of a given message. The state of the DFSM is used to govern the allowable sequence of commands for a protocol. Each state has a number of transitions where, each transition has a single input command and a sequence of output commands of any length (including zero).

Figure 2 illustrates how a DFSM may be used to model the protocol for the target LDAP server functionality described in Section 3. This model allows for an LDAP client to issue search and add requests only after it is properly bound. The model is simple and as such it does not completely describe rich LDAP server behaviour: Bind requests are assumed to be valid and result in a bound connection. A search request always results in a single search result entry followed by a search result done. A more powerful protocol model would be required to express richer behaviour, such as, unsuccessful bind requests and zero or more search result entries for a search request.

For the purpose of our work, we defined ECTO, an XML-based language to represent the states, transitions

```

<state id="bound" initState="no">
  <transition>
    <incommand idref="searchReq" />
    <nextstate idref="bound" />
    <outcommand idref="searchRes" />
    <outcommand idref="searchResDone" />
  </transition>
  <transition>
    <incommand idref="addReq" />
    <nextstate idref="bound" />
    <outcommand idref="addRes" />
  </transition>
</state>
<command id="searchReq">
  LDAP Search Request</command>
<command id="searchRes">
  LDAP Search Response</command>
<command id="searchResDone">
  LDAP Search Response Done</command>
<command id="addReq">
  LDAP Add Request</command>
<command id="addRes">
  LDAP Add Response</command>

```

Listing 1. ECTO Specification of LDAP State Machine.

and commands of a protocol. The following code sample illustrates how the **Bound** state and associated transitions and commands may be expressed in ECTO as given in Listing 1.

Each state node contains many transition nodes, which each describe a single transition in the DFSM using an incommand, nextstate and any number of outcommand nodes, respectively.

Message Structure. The DFSM described above can approximate the behaviour of an enterprise protocol. In order to provide an emulation of that interaction behaviour additional information is required regarding the content of the messages that will be exchanged between the endpoints. Consider the following text representation of an LDAP Search Request:

```

LDAP Search Request
Message ID: 6
LDAP Search Request Protocol Op
Base DN: ou=Corporate ,o=Swinburne ,c=AU
Scope: 0 (baseObject)

```

It is straightforward to identify that the command of this example message is an LDAP Search Request. However, there is additional information in the message that requires consideration in order to generate an appropriate output message. For example, an LDAP Search Request with a certain Message ID expects that the response to that search request has a matching Message ID. A Message ID may be any integer between 0 and $2^{31}-1$, attempting to model this

using an DFSM would result in a very large DFSM. To help overcome this limitation of DFSM's, we have chosen to define a message as a sequence of non-overlapping *regions* drawing on terminology used in recent work on reverse engineering protocols [10] .

A region is further decomposed into two sub-categories, the *command* region and the *field* region. A command region denotes the command of the given message, for simplicity there may be only one command present in any individual message. A field region is a key/value pairing which is used to represent parameters of a command and their associated values. To illustrate the use of a field region consider the LDAP Search Request example. The Message ID becomes the *key* for a field and the *value* of that field becomes 6.

Through the use of command and field regions we define the *expected message formats* for an enterprise protocol. An expected message format describes the expected fields of a message based on the command of that message. We assume the existence of a mechanism for eliciting the command of a message and the subsequent field values at run-time.

4.2. Enterprise Endpoints

An enterprise endpoint often refers to its local data in responding to a request. For example, an enterprise endpoint processing an *LDAP Search Request* needs to respond with the appropriate data found in its directory. The typical response to a *LDAP Search Request* is a *LDAP Search Result Entry* as illustrated here:

```

LDAP Search Result Entry
Message ID: 8
dn: ou=Corporate ,o=Swinburne ,c=AU
CN: cn=Cameron Hine
CN: cn=Jean-Guy Schneider
CN: cn=Jun Han
CN: cn=Steve Versteeg

```

The DFSM model we have described in this work describes that this LDAP Search Result Entry command is the result of a LDAP Search Request. The expected message format enables the Message ID: 8 and dn: ou=Corporate,o=Swinburne,c=AU of the corresponding request to be preserved and subsequently attached to this response. However, the CN values depend on the data available on the endpoint being issued the request. Hence, some representation of this data is required in order to build a valid response.

We model endpoint data as a single *datapool* shared by all emulated endpoints. The shared datapool approach allows us to minimize the amount of data required to be stored for an emulation, as any data which is common to multiple emulated endpoints needs only to be stored once. To achieve variation in the data between endpoints, we assign different start and end indices for each emulated endpoint which mark

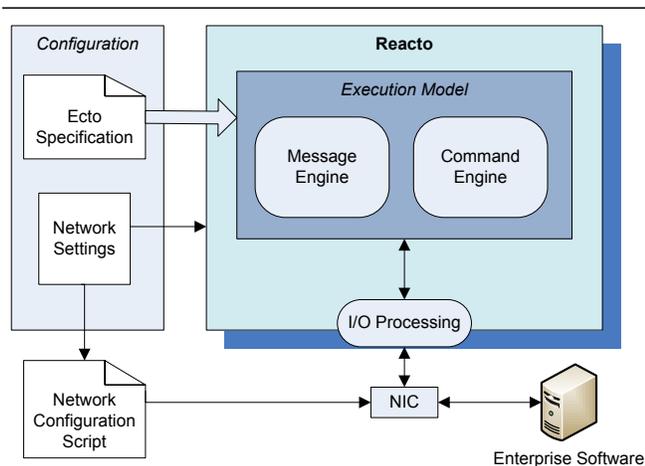


Figure 3. Architecture of REACTO

the section of the datapool that an endpoint has access to. This ensures that emulated endpoints do not appear identical to one another.

5. Reacto: Prototype Emulation Environment

As a proof of concept of the approach introduced in the previous sections, we have implemented REACTO, a prototype reactive emulation environment. The high-level architecture of REACTO is shown in Figure 3. In this section, we will further discuss the chosen architecture and highlight some of the design decisions we made.

5.1. Configuration

It is our goal that REACTO supports many different environment scenarios. By using abstract models and an execution model to define the behaviour of endpoint systems, REACTO can inherently emulate a range of enterprise endpoint services. REACTO is configured by an ECTO specification that contains the description of the enterprise protocols and endpoint models that will be used to drive the emulation behaviour. Based on this specification, REACTO will construct a corresponding *execution model* that comprises a *Message Engine* and a *Command Engine*, respectively. REACTO also needs network settings that define the range of ports that the *Message Engine* will listen on. Please note that the current prototype does not support the emulation of multiple protocols within a single execution, but such functionality may be addressed in future work.

5.2. Capturing network addresses

To receive requests intended for multiple endpoint systems, REACTO captures the network addresses of the end-

point systems it intends to emulate. This allows the enterprise software to send out requests in the same way as it would in a real deployment environment, unaware that all requests are in fact all being received by the emulation environment. REACTO accomplishes this capture through the use of the *Ethernet aliasing* feature of Linux. The IP addresses of the endpoint systems being emulated are mapped to a single network interface card (NIC) on the machine running REACTO.

This approach allows REACTO to receive requests originally intended for the endpoint systems being emulated, without needing to modify any enterprise software that is tested using REACTO. It also comes with the advantage of leaving the original intended IP address of packets intact, allowing REACTO to map requests based on IP addresses to the appropriate emulated endpoint systems.

An alternative approach we considered was using *Network Address Translation (NAT)* to map requests intended for many different IP addresses to a single physical machine. However, we found NAT to be unsuitable as it manipulates the IP addresses of packets, which REACTO relies on to identify the original intended destination of requests.

5.3. Execution Model

A run-time environment requires the ability to generate and send a response upon receipt of a request according to the protocol and endpoint models supplied by enterprise software developers. REACTO uses what we term an *execution model* to achieve this end. The execution model translates the protocol and endpoint model specifications into appropriate computational structures and algorithms, which obey the rules defined in the models, and can be used to generate responses to requests.

The execution model of REACTO consists of two key components, a *Message Engine* and a *Command Engine*. The *Message Engine* contains the structures that define the *expected message formats* for services being emulated. When a request is received by REACTO a string representation of that request is passed to the *Message Engine*. By matching the string representation of a message against the *expected message formats* the *Message Engine* can discover what *command* is to be issued as well as the values for any parameters. This information can then be passed along for further request processing as a *Message* structure, representing the internal base unit of communication that contains a single *command* and any number of *fields*.

The *Command Engine* contains the DFMS structures for the services being emulated. When passed a request *Message* for processing, it examines the *command* being issued by the request, updates the DFMS for the appropriate emulated endpoint system and returns any output *messages* that need to be issued as responses.

The output *messages* contain information about the required output *command* and the original values of any *fields*. In some cases this is all that is needed to generate a response. Otherwise, some endpoint dependent *data* needs to be inserted into an output *message*, e.g., the result of an LDAP Search operation. The *Message Engine* has access to the *datapool* which holds the data that will be used to populate the endpoint dependent data. When an output *message* is received by the *Message Engine*, any required *fields* for the output *command* are filled with the corresponding values from the original request *message*. If there is no corresponding value, the *Message Engine* will fill the output *field* using appropriate endpoint data from the *datapool*.

5.4. Run-time Design

In order to address scalability, we have chosen to use a multi-threaded architecture for REACTO as is illustrated in Figure 4. This choice was also motivated by the desire to utilize machines with multi-core architectures and scale the environment beyond the limits of single-core machines. In the following, we will high-light some of the important aspects of the underlying architecture.

I/O Processing: There are 4 threads responsible for network I/O in REACTO, the ConnectionAcceptor, SocketListManager, ReadyQueueWorker, and OutMsgWorker. The ConnectionAcceptor listens on a single port on the emulation machine waiting for new connection requests. When a new connection is detected, a socket is constructed and placed onto the socketList buffer. This newly created socket is used for all future communication between the enterprise software connected to REACTO and the emulated endpoint system which it intended to connect to. The socket structure keeps track of the intended destination for the connection request which acts as a unique identifier, mapping requests arriving on sockets to the correct emulated endpoint.

The SocketListManager takes care of event-based I/O by continuously checking the socketList for available data. If a socket is found to have data available and it is not already present on the readyQueue, it is placed onto the readyQueue. The ReadyQueueWorker consumes sockets from the readyQueue reading data for a single message from each socket and passing it to the *Message Engine*. Every socket on the readyQueue is processed with equal priority, regardless of the queue size and amount of data available on each socket. The *Message Engine* returns a *Message* object that contains all the information needed for further processing. The OutMsgWorker consumes output messages from the outMsgQueue, which are then passed through the *Message Engine*, turning them into raw data which can be sent as a response. These responses are sent using the same socket which the original request arrived on.

This I/O strategy ensures that in a stable emulation, where no new sockets are being created, every socket gets an approximately equal opportunity for using the network throughput of the machine, independent of the load characteristics of the system. For example, consider the situation where requests R_1 and R_2 arrive for endpoint E_1 in quick succession. The SocketListManager notices that E_1 has data available and places it onto the readyQueue. The ReadyQueueWorker reads R_1 from the socket and continues processing. While request R_1 is undergoing further processing, request R_3 arrives for endpoint E_2 and subsequently the SocketListManager places the appropriate socket onto readyQueue. This situation will result in request R_3 being processed before request R_2 , even though request R_2 arrived before R_3 .

The design of the event-based I/O model ensures that no new threads need to be created in order to handle new connections. Doing so would significantly limit scalability as having many thousands of threads active on one machine would result in computationally expensive context switching between threads. The I/O strategy we have chosen results in relatively stable throughput despite the number of emulated systems, as will be shown in Section 7.

Message Engine: The REACTO message engine performs two key tasks: (i) translating a raw request into an input message structure which can be used for processing by other REACTO components, and (ii) translating an output message structure (issued as a response to an input message) into a raw data output which can be understood by the enterprise software that originally issued the request.

The first operation the message engine must perform when it receives a request is to decode the (raw) data and turn it into a text string which fully describes the corresponding message. Once the text representation has been retrieved from the data, the message engine can match that text against the *expected message formats* and using the approach described in Section 5.3, a message structure is created. This message structure encapsulates the input command as well as the fields of the message that was received.

The message engine is also responsible for generating raw data (as a response to a request) from an output message. The first step is to identify any fields that need to be filled using the datapool. This is done by examining the *expected message format* of the output message and checking for any expected fields that do not already have values. These fields are filled with data from the datapool, using the start and end indices assigned to the emulated endpoint system. Finally, the output message is encoded in the same format which the original request arrived in.

Command Engine: The command engine is responsible for managing the DFSM for every emulated endpoint. Each endpoint system has a unique network address and a single current state. The InMsgWorker passes input messages to the

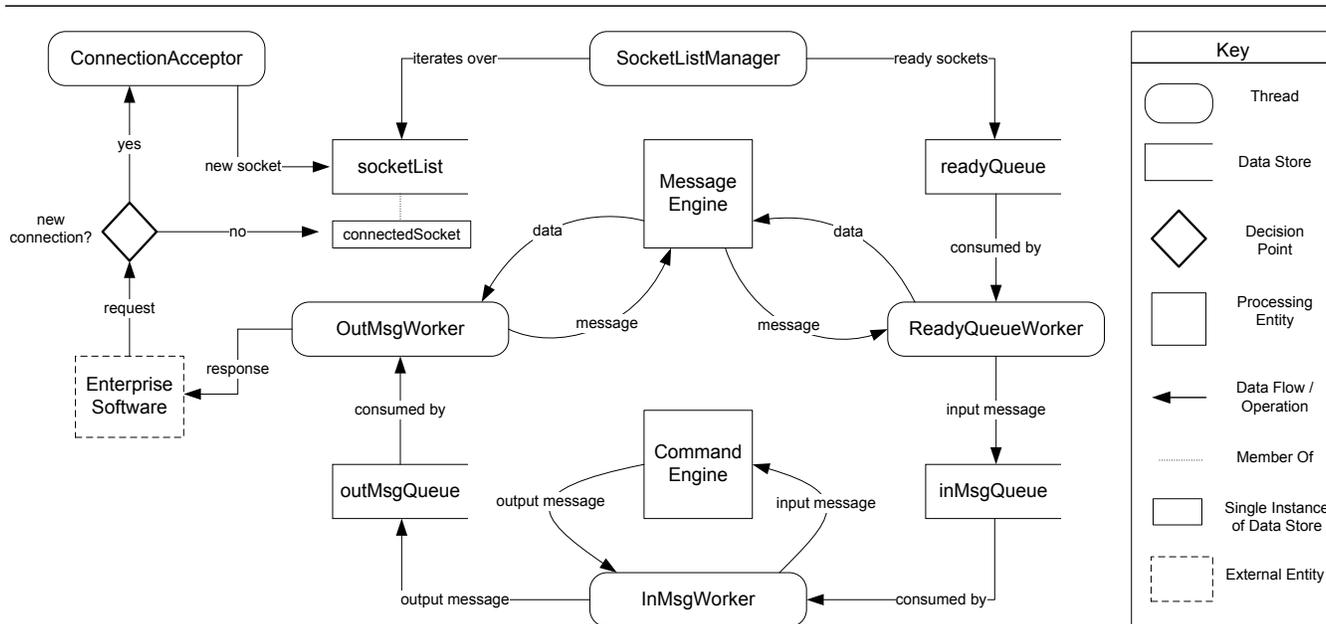


Figure 4. Run-time Design of REACTO.

command engine which uses the socket information in the message to retrieve the current state of the corresponding endpoint. It then applies the command to the DFSM and encodes the output command in the corresponding output message, containing the output command as well as the fields of the original input message.

6. Practical Validation of Approach

The intended goal of the emulation environment is to enable non-functional testing of enterprise class software systems. To evaluate whether the proposed environment may indeed help in achieving this goal, we conducted preliminary validation with an existing enterprise identity management suite: CA Identity Manager Release 12.0 [7]. We tested whether the REACTO prototype could successfully interact with *Identity Manager*, and whether the interaction could be scaled.

We conducted a test where REACTO was used to emulate 10,000 LDAP endpoints on a single physical machine. CA *Identity Manager* then connected to the REACTO emulated endpoints to conduct some basic identity management tasks. Three tasks were undertaken:

- Acquire Endpoint: involves binding to the corresponding endpoint and then performing a search which retrieves the objects at the top level of the directory tree.
- Explore Endpoint: involves a hierarchical search which traverses the directory tree looking for objects which

Identity Manager can “manage”. The value of the search requests sent by *Identity Manager* depend directly upon the previous responses from the LDAP endpoint.

- Add Account: consists of an add request, containing the account information of a new user, being sent to the endpoint. The endpoint needs to respond back to indicate whether the add succeeded.

The CA Provisioning Manager GUI tool was used to get *Identity Manager* to perform each of these three identity management tasks on REACTO emulated endpoints. *Identity Manager* was able to execute each of these tasks in the same way as if it was interacting with a real LDAP server. For Explore Endpoint, REACTO returned the search results for a shallow tree. In the case of Add Account, REACTO reported back to *Identity Manager* that the add had been successful, without actually performing an update to the data-store.

We then set up an automated test, using an Apache JMeter script, in which a single instance of CA *Identity Manager* connected to 10,000 REACTO emulated endpoints. *Identity Manager* performed an Add Account request at each emulated endpoint. For each request, a Reacto emulated endpoint processed the request and sent an add success response to *Identity Manager*. Throughout this testing, REACTO consumed at most 32 Megabytes of main memory.

These tests show that the concept that Reacto emulated endpoints can successfully interact with an enterprise system product, such as CA Identity Manager, and that the emulation can be efficiently scaled.

7. Scalability Study

The prototype emulation environment REACTO was also used to conduct some preliminary validation of the scalability of the approach. We used REACTO to emulate differing numbers of endpoint systems, each emulating the search functionality of an LDAP server. The primary goal was to investigate the scalability of the approach as well as our prototype implementation. In particular, we were interested in the relationship between the number of emulated endpoints and the resulting network throughput. This relationship is an important measure to assess the scalability of both, our approach as well as the REACTO prototype. If the throughput decreases significantly when the number of emulated endpoints increases, then we can conclude that scalability is limited.

Furthermore, we choose to investigate the performance of different stages of request processing in order to identify which stages of request processing are computationally intense and likely to become a performance bottleneck. It should be noted that the timing data we collected did not allow for fine-grained analysis of the network input and output characteristics of REACTO. For example, we did not measure the time it takes for individual requests to be processed from the original time the request was generated – only the total number of requests that were able to be processed in a specified amount of time was considered.

7.1. Environment

The environment used to obtain the results consists of two machines, one running REACTO to provide the emulation of LDAP search functionality, the other running *dxsoak*, an LDAP load generation tool specifically tailored towards LDAP directory benchmarking. REACTO was running on a Dell Optiplex GX620 running the Gentoo Linux distribution with 2GB of RAM whereas *dxsoak* was running on a Dell Latitude D620 running Windows XP Professional Service Pack 2, also with 2GB of RAM. Each machine had a single 1 Gigabit Ethernet card, connected directly using a CAT-5 cable.

In order for the emulation machine to be able to receive LDAP search requests for every emulated system, the REACTO machine was configured so that its NIC had 10,000 IP addresses: 128.0.1.(1-250), 128.0.2.(1-250), ..., and 128.0.40.(1-250).

7.2. Throughput Characteristics

In order to examine the throughput characteristics of REACTO, we measured the total number of received and transmitted search request at the machine running REACTO when emulating 1, 50, 250, 1000, 2000, 5000, 8000 and 10,000 endpoints under maximum load from *dxsoak* for

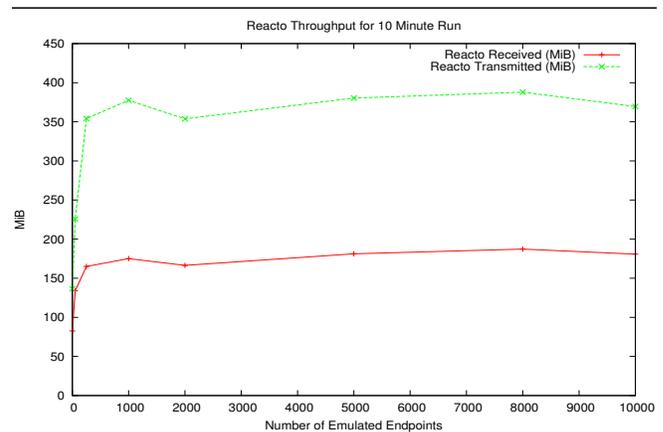


Figure 5. Reacto Throughput Characteristics

a period of 10 minutes each. We also measured the total number of searches completed by *dxsoak* in 10 minutes for 1, 50 and 250 emulated systems.

The load generated by *dxsoak* can be characterized as many small repeated search requests. On average, approximately 4483 LDAP searches were completed for every transmitted Mebibyte (MiB).¹ When emulating a single endpoint, we got a result of 667,333 searches completed in the 10 minute period, which is approximately 1112 LDAP searches being completed every second. When we increased the number of emulated endpoints to 250, we achieved 1,080,804 searches completed, which corresponds to approximately 2227 LDAP searches being completed every second. As we increased the number of emulated endpoints, only minor variations in the total throughput was observed.

The throughput characteristics of REACTO is shown in Figure 5. It clearly shows that the total throughput of REACTO does not significantly degrade with an increased number of emulated endpoints and that REACTO can indeed emulate 10,000 enterprise endpoint running LDAP search. However, in a system where bandwidth is shared equally between connections as the number of emulated systems increases, the throughput available for each of those systems drops according to the relationship $T_s = \frac{T_t}{E_s}$ where T_s is the available throughput for an individual emulated system, T_t the total throughput available to the emulation, and E_s the number of emulated systems. This means that even though we are able to emulate 10,000 enterprise systems, the available throughput for each of those systems is limited.

7.3. Request Processing Analysis

To better understand how well the REACTO prototype performed at runtime we inserted timing code which reported

1. http://www.iec.ch/zone/si/si_bytes.htm

Stage	Percentage
Read data from socket	10.31%
Decode and create message structure	48.63%
Create output message	12.73%
Encode output message	19.56%
Send response on socket	8.06%

Table 1. Request Processing

on the amount of time it took to complete the various stages of request processing. The results are summarized in Table 1.

The results indicate that almost half of the total amount of time it takes to process a request is spent decoding the request and constructing the input message structure. This step is required to easily enable human analysis of messages as they pass through the emulation. Optimizing decoding and encoding in future prototypes is expected to result in a significant increase in performance. Our results also illustrate the effectiveness of the I/O strategy used by REACTO. On average, 18.37% of total processing time was spent reading and writing on the socket. This means that the current prototype is not bound to network I/O and future work can focus on improving the other operations required to process requests.

8. Related Work

Over the past years, a substantial amount of research into testing distributed software systems was conducted. In 1999, Ghosh and Mathur [8] provided a broad outline of the issues involved in this area, followed by Canfora and Di Penta that presented similar issues from a service-centric perspective [4]. Both works highlight similar issues unique to testing distributed software systems. In particular, the issue of how to test a software component which requires interaction with another distributed component for operation is relevant to our work. The emulation approach we propose can help address this issue by providing a communication partner capable of representing a distributed component for the purposes of testing. In 2004, Denaro, Polini and Emmerich presented an approach to conducting early performance testing of distributed software applications [6]. The method proposed made use of stubs to mimic expected component behaviour so that performance testing could take place before the actual components were available. The stubs were not required to model complex interaction behaviour as this was not needed to record valid performance testing results. We suggest that in some cases complex interaction behaviour is required to conduct such testing. We also suggest that a suitably powerful model, emulated by an execution environment, is capable of representing such complex interaction behaviour.

The models of our emulation environment need to describe the interaction behaviour of the corresponding endpoint systems. Various formalisms have been proposed to describe this interaction behaviour. State based models have been widely used to describe communication protocols [2], [5], [11]. These models express interaction through states and transitions, incoming and outgoing messages represented by transitions. Calculi such as Milner's CCS [13] and the π -calculus [14] capture issues of communication and concurrency using a small set of algebraic primitives. These models express interaction through a notion of processes that are connected to one another other along *channels* that facilitate interactions by passing messages between processes. The required model must capture the communication protocol issues of interactions which the state-based and calculi-based approaches can help to address. The model must also deal with the underlying issues of request processing so that a request received by the emulation can be properly processed and responded to at runtime. This requires a lower level of abstraction for individual messages coming in and going out of the emulation. The model proposed in this paper is preliminary, the state machine captures aspects of the communication and the datapool is used to inject data into individual messages as required.

Network emulation is an established approach to investigate and test distributed communication protocols. Significant efforts in this area include the Virtual InterNetwork Testbed (VINT) project [3] and the ModelNet network emulator [17]. The focus of research in this area is on design, correctness, and efficiency of lower-level communication protocol implementations. Our focus on the other hand, is on emulating application-layer protocols and the endpoint systems which use them, so that we can enable non-functional testing of enterprise software which rely on those systems for operation. Similarly to our goals, some network emulation tools are able to scale to very large computer networks [17]. These efforts however, make use of cluster architectures consisting of many physical machines, we intend to emulate a scalable number of endpoints using a limited number of physical machines.

9. Conclusions and Future Work

Conducting non-functional testing on enterprise software that relies on a large number of enterprise systems for standard operation is a challenging task. It is generally not practical to physically set up and maintain the large network of machines that would enable testing of this class of software in realistic deployment environments. In this paper, we investigated an endpoint emulation approach as a means to enabling non-functional testing for these kinds of systems. The fundamental idea behind the emulation approach is to use lightweight models of endpoint interaction behaviour to synthesize a scalable and interactive emulation of an

enterprise environment. It is expected that this emulation may be used in place of a real enterprise environment to conduct non-functional testing activities.

In this preliminary work, we use a deterministic finite state machine to model a subset of LDAPv3 protocol and a shared datapool to model local endpoint data. In an emulation the datapool is used whenever endpoint contextual data is required to build a valid response. We demonstrated the scalability of the models and the approach through a prototype tool REACTO. This tool was shown to emulate up to 10,000 LDAP servers simultaneously on a single physical machine. The practicality of the approach was validated by experiments with an enterprise class commercial software system: CA's Identity Manager. *Identity Manager* was connected to REACTO and used to perform some common tasks which required interaction with the emulated endpoint systems. It was shown that the emulation REACTO provided was indeed adequate for *Identity Manager* to complete these tasks.

This work has uncovered a number of open questions that we intend to address in the future: (i) In some cases the contents of the emulated responses directly effect future interactions. In these cases the content of those messages needs careful consideration; the message content must be valid enough to continue the interaction. (ii) One of the shortcomings of the current models and prototype is that it does not support *modification* operations on the emulated endpoints datapool. This functionality is of importance to any enterprise software which expects data on endpoints to be manipulated as the result of a previous command. (iii) Real enterprise environments exhibit a range of temporal and behavioural characteristics. For instance, endpoint response time will vary depending on endpoint system load and complexity of the processing the corresponding request. Also, in some cases endpoints may be faulty and respond to requests in unexpected ways. These characteristics and others may cause faults or performance degradation in the enterprise software systems we intend to test. Hence, future work will investigate approaches to modeling and integrating these unpredictable characteristics into a reactive emulation environment.

Acknowledgments

We would like to thank CA Labs for their extensive support of this work, Tim Ebringer for his valuable discussions and suggestions, and the anonymous reviewers for their feedback.

References

- [1] G. V. Bochmann and C. A. Sunshine. Formal Methods in Communication Protocol Design. *IEEE Transactions on Communications*, 28(4):624–631, 1980.
- [2] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, April 1983.
- [3] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *Computer*, 33(5):59–67, May 2000.
- [4] G. Canfora and M. D. Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8(2):10–17, March-April 2006.
- [5] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE01)*, pages 109–120, New York, NY, USA, 2001. ACM Press.
- [6] G. Denaro, A. Polini, and W. Emmerich. Early Performance Testing of Distributed Software Applications. *SIGSOFT Software Engineering Notes*, 29(1):94–103, 2004.
- [7] M. Gardiner. CA Identity Manager, November 2006. White Paper on CA Identity Manager.
- [8] S. Ghosh and A. P. Mathur. Issues in Testing Distributed Component-Based Systems. In *In First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.
- [9] HP. HP LoadRunner Software Data Sheet, May 2007.
- [10] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: an automated script generation tool for honeyd. In *ACSA 2005, 21st Annual Computer Security Applications Conference*, Tucson, USA, December 2005.
- [11] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
- [12] S. Microsystems. SLAMD Distributed Load Generation Engine – Release Notes, April 2006. Version 2.0.0 Alpha 1.
- [13] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [14] R. Milner. *Communicating and Mobile Systems: The π -Calculus*. Cambridge University Press, New York, NY, USA, 1999.
- [15] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511 (Proposed Standard), June 2006.
- [16] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.
- [17] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.
- [18] J. Watson. VirtualBox: Bits and Bytes Masquerading as Machines. *Linux Journal*, 2008(166):1, February 2008.