Service-Oriented Performance Modeling the MULE Enterprise Service Bus (ESB) Loan Broker Application

Paul Brebner
NICTA, Canberra Research Laboratory
Computer Sciences Laboratory, RSISE, ANU
Canberra, Australia
e-mail: Paul.Brebner@nicta.com.au

***Abstract*—Since 2006 NICTA has been developing and trialing Service-Oriented Performance Modeling (SOPM), a method and tool support for performance modeling of large-scale heterogeneous Service Oriented Architectures (SOAs). This technology enables software architects to rapidly build performance models of SOAs directly in terms of service compositions. Enterprise Service Buses (ESBs) are an increasingly common style of SOA infrastructure and implementation technology that we have encountered and modeled in e-Government SOA projects. In this paper we show the application of our SOPM approach to the MULE Enterprise Service Bus Loan Broker application in a laboratory context. We give a high-level outline of the SOPM method, and introduce the MULE ESB and Loan Broker application. We describe how a SOPM of the Loan Broker application is built in terms of application business-logic services and MULE infrastructure service components, and parameterized with measurements from an experimental test-bed. We demonstrate the validity of the approach in an initial scenario, and then explore the modeling of alternative deployment and application scenarios.**

Service-Oriented Performance Modeling; Enterprise Service Bus; Performance; Scalability

I. INTRODUCTION

Since 2006 NICTA has been developing and trialing Service-Oriented Performance Modeling (SOPM), a method and tool support for performance modeling of large-scale heterogeneous Service Oriented Architectures (SOAs). It is vital to understand SOA performance and scalability characteristics before system switch-on to prevent meltdown, because SOAs: often depend on legacy or 3rd party services with limited resources; demand for new services is hard to predict and potentially unbounded (particularly if available as "open" services via the internet); and services are increasingly used for mission critical applications.

Load testing is a common strategy used to measure the capacity of traditional software systems, but it is often technically difficult, expensive or impossible to adequately load test SOA applications end-to-end. Barriers to load-testing SOAs include: testing across organizational boundaries; security requirements; lack of tools and skills; the presence of resources that are shared with other organizations and/or production systems; the use of services provided by other organizations; load testing may be perceived as a denial of service attack; Service Level Agreements (SLAs) may impose restrictions on use; and there may be a cost to use a service or other infrastructure such as networks. By the time that integration testing is conducted on a production ready system it is inevitably too late and too expensive to radically change the software architecture to address performance and scalability deficiencies. It is therefore valuable to predict the performance implications of architectural alternatives for SOAs as early as possible during the development lifecycle.

We have developed our own modeling tool in order to enable rapid GUI-based construction, use, and maintenance of SOA performance models by software architects. The tool supports direct modeling of SOAs in terms that architects are familiar with (composite and simple services, workflows, workloads, servers), and using a model-driven approach automatically generates a run-time model for execution by a custom discrete-event simulation engine. The simulation is dynamic and various parameters can be changed and metrics graphed while it is running. The performance and scalability of alternative scenarios and parameter values can be easily compared. The tool directly supports modeling complex composite services (we have built models with 10s of services, servers, and workloads, and 100s of workflow steps), and the model components are themselves reusable and composable (i.e. composite services, simple services, and workloads are all interchangeable in the model). Metrics are computed and can be graphed for all model components, including workloads, services, and servers. The ability to predict metrics for services is critical for SOA performance prediction, as they are necessary for understanding SLAs between service providers and consumers, and resources (e.g. threads, CPU, memory requirements, etc).

In parallel to the tool we have also developed and tested a method to apply SOPM to a wide-range of government and non-government projects and architectural styles, including: whole-of-government services; web-based systems; n-tier architectures; search services; model-driven architectures; different SOA and orchestration styles; networks and streaming data; and alternative resource scheduling (including server virtualization and cloud computing). Some open issues we have recently identified for SOA performance modeling include better integration with run-time environments and business processes, and more accurate modeling of service orchestration mechanisms, including Enterprise Service Buses (ESBs) [4]. One of the initial and ongoing

978-0-7695-3784-9/09 $26.00 © 2009 IEEE
DOI 10.1109/SEAA.2009.57

applications of SOPM was to a whole-of-government security service [3] which was implemented in Microsoft Windows Communication Foundation (WCF) "ESB" [17]. However, the ESB infrastructure wasn't modeled explicitly in this engagement, and another ESB example was needed to explore the suitability of the approach for modeling ESB components explicitly. This paper focuses on initial experiments conducted in our laboratory with the application of the SOPM approach to an Open Source MULE Enterprise Service Bus Application.

The goal of the paper is to determine if performance models of ESB applications can be easily constructed using the approach. We want to see if the models correspond closely to the ESB infrastructure and applications modeled, as this will enable rapid construction, ease of understanding and modification, and facilitate use and maintainability of the models. Other issues to investigate are as follows. Can performance data (response times) be obtained from a running ESB application in order to parameterize the model? How closely does this data map to the model components? How accurate are the resulting predictions (throughput, etc)? Can other relevant metrics be predicted (E.g. timeouts, concurrency, SLAs)? Can alternative scenarios (e.g. deployment alternatives) be modeled?

The structure of the paper is as follows. Section II introduces the MULE ESB and the Loan Broker ESB application. Section III describes the SOPM of the Loan Broker Application. Section IV describes the configuration of the test environment used to gather performance data to parameterize the model and validate the initial scenario. Section V is the main body of the paper in which the model is used and adapted for a number of different deployment scenarios, starting from the simplest possible use of the default configuration of the Loan Broker application with no modifications, and progressing to more realistic deployment and usage scenarios. Section VI summarizes previous work, and we conclude and suggest future work in Section VII.

II. MULE ESB & LOAN BROKER APPLICATION

MULE [5] is a java-based Enterprise Service Bus framework designed to ease the task of integrating multiple heterogeneous enterprise components by separating the business logic layer from the messaging layer. It supports integration and reuse by enabling service components (which implement business logic) to be used in multiple different ways, by loosely coupling them together using a variety of standard MULE messaging, transport, routing, connector, adaptor, aggregator, filter, security and transformer components acting as gateways and mediators. Orchestration is managed by the ESB, and different topologies are supported for both applications and the ESB itself.

The Loan Broker application is a widely used SOA/ESB application example, and is well documented in the Enterprise Integration Patterns book [9]. There are a number of different versions of this application available for MULE [15]. The Loan Broker ESB Example implements a full ESB architecture based on Java Message Service (JMS) messaging, the Loan Broker ESN versions implement both synchronous and asynchronous point-to-point routing, and the Loan Broker BPM uses a Business Process Engine to orchestrate the requests. In this paper we use the Loan Broker ESB example [16] as the focus of the experiments and modeling, but intend to extend the work to the other examples in the future. The Loan Broker ESB example demonstrates using HTTP/REST, Enterprise Java Beans (EJB), JavaBeans, and SOAP Web Services. It uses a JMS message bus to provide a common messaging channel between the different components and applications. Fig. 1 [16] shows the design, including the workflow (progressing from step 1 to 10), the message endpoints, and components types. The MuleClient represents the remote consumer of the LoanBroker service. LoanBroker, CA Gateway, Lender Gateway, and Banking Gateway are all MULE components. The circles represent components or services that are accessed via the MULE Gateway components.

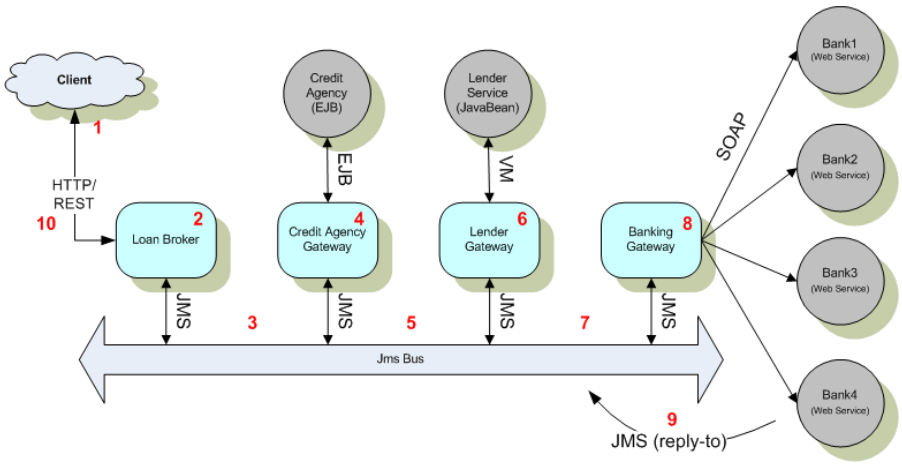

Figure 1. Loan Broker ESB Application design [16]

III. PERFORMANCE MODELLING THE MULE ESB LOAN BROKER APPLICATION

NICTA's SOPM technology is a methodology and tool support to enable the performance modeling of architectures in terms of Composite and Simple Services.

Composite Services model business processes and services which consume other services, and are represented as a workflow consisting of an ordered series of steps, where each step is a call to another service. By default all workflow steps are executed, and service calls are synchronous and sequential. For more complex service orchestrations we can model choice of service calls (based on probabilities), and concurrent or asynchronous calls. Services that are implemented by components deployed to multiple servers are modeled as Composites, even if the components aren't strictly services.

Simple Services model services which cannot or do not need to be further decomposed, typically because they are hosted externally by 3rd parties, because no detailed implementation information is available for them (e.g. legacy applications wrapped as services), or because they are completely hosted on a single physical resource so there is no need to model them at a finer granularity to correctly predict scalability. In SOPM models, simple services are deployed to a single physical server. Servers are parameterized with a capacity corresponding to the number of CPUs (equivalent to cores for multi-core systems).

For a complete SOPM, the external use of the services is modeled using workloads. Workloads are described as workflows, representing the use of services by an external application. Workloads are parameterized with the number of concurrent users, the arrival interval (average time between user requests), and the arrival distribution (e.g. constant or Poisson).

Finally, the service times are parameterized with measured performance data from an unloaded system. Simple services either have a default typical time, or the time for each service call can be overridden or set in the workflow steps. This is used if a service takes a different amount of time depending on how it is invoked (e.g. different protocols, methods, parameters, payloads, etc). Composite service times (and other metrics such as concurrency and throughput, and other metrics for workloads, services and servers) are computed by the simulation engine given the workload and service times, and server capacities. We now show the application of SOPM to the Loan Broker example.

The first step in the SOPM method is the identification of services, both composite and simple. This is typically done from architecture documentation, but needs to be checked and possibly modified from run-time information, as performance data needs to be correctly mapped to the modeled services. The Loan Broker ESB application (Fig. 1) can be modeled as follows. The six "external" services invoked by the application (Credit Agency, Lender Service, and Bank1 to Bank4 Web Services) do not call other services, and do not have any further implementation details available. We therefore model them as simple services. The remaining four MULE components (Loan Broker, Credit Agency Gateway, Lender Gateway, and Banking Gateway) are modeled as composite services.

We initially modeled only one Bank web service to simplify the experiment (and only used loan requests that resulted in quotes from one bank service). Time spent in the MULE services/gateway components was modeled with a single MULE service for simplicity (with different times specified for each component), rather than creating a unique service for each MULE component. The MULE service was deployed to a single MULE server (with 2 CPUs corresponding to the dual-core server used for testing). This MULE service is introduced purely as a modeling artifact in order to correctly model time spent in the MULE components themselves. The other "external" simple services were also initially deployed to the same server (the default configuration for the application). Messaging between components can also be modeled with our approach (e.g. HTTP/REST, JMS, EJB, JVM, SOAP), but preliminary performance profiling of the application indicated that messaging times were (a) very short compared with the service times (particularly for JMS, EJB, and JVM), and (b) difficult to measure accurately with sufficient precision. We have therefore chosen not to model them in this paper.

Finally, the Mule Client is modeled as a workload calling the Loan Broker composite service. The screenshot (Fig. 2) shows the initial Loan Broker SOPM with one banking service only. This screenshot shows the model components and relationships. Starting in the left-hand top corner, the Mule Client is modeled as a workload (oval) which calls the LoanBroker Composite service (rectangle with round corners). Composite services have 1 or more steps (shown as ovals below the composite component, which model the sequence of steps) which call other services. Simple services are represented as rectangles, and are initially deployed to a single server (circle at top right). Lines from workloads to services, or services to services represent service calls, and lines from simple services to servers model a deployment relationship. Times (in ms) are explained next.

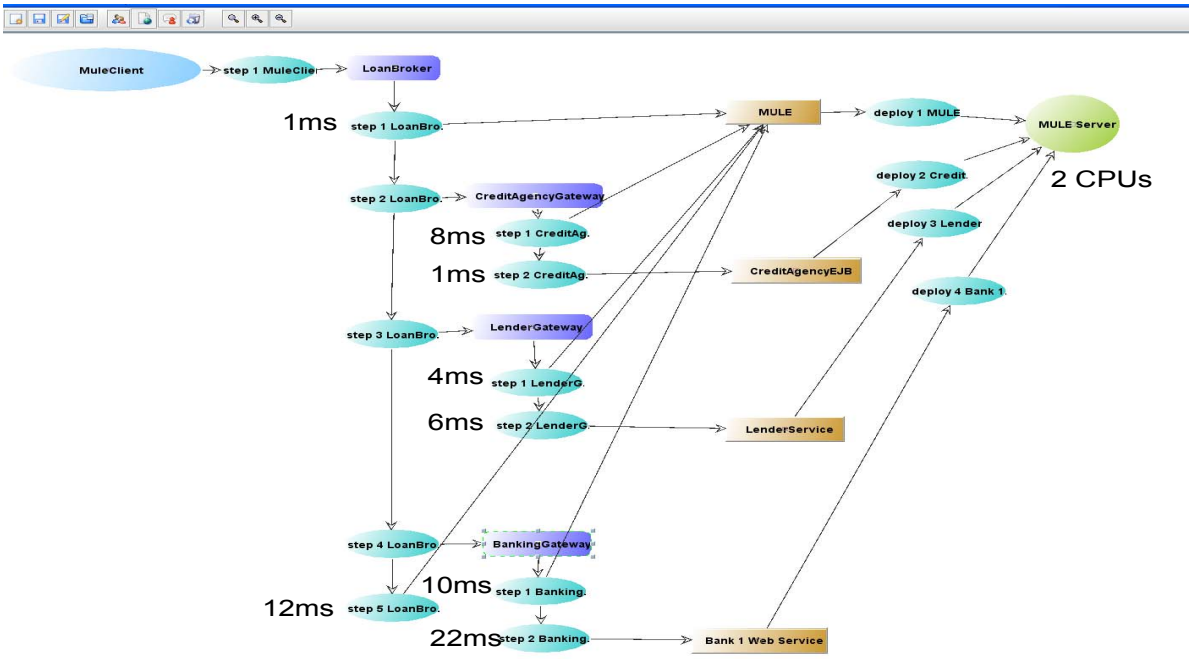

Figure 2. Screenshot of initial Service-Oriented Performance Model with one Bank Service

IV. EXPERIMENTAL TEST CONFIGURATION AND MODEL PARAMETERIZATION

We set up a test environment in order to obtain performance measurements to parameterize the model, and to obtain load testing results to validate the model. A 2-tier architecture was configured on two machines. A JMeter client (used to invoke the Loan Broker service via HTTP/REST) ran on a dual core server. Another dual core desktop server (AMD Athlon 64X2 4600+ 2.41GHz, 4GB RAM, Windows XP Service Pack 2, Mule 2.1, and Java 6), ran the Loan Broker application. The CreditAgencyEJB was deployed using the option as a real Enterprise Java Bean (EJB) as this is more representative of an Enterprise deployment, and facilitates modeling alternative deployment scenarios. Windows XP was configured to allow an arbitrary number of incoming HTTP requests a second to enable sufficient loads. In order to produce accurate performance predictions a SOPM needs to be calibrated with performance data from an unloaded system. Measured or assumed parameters for the model include service response times, number of users and arrival intervals for workloads, and server capacities. Metrics computed by the simulation engine for all components include response time, throughput, and concurrency, and Utilization for servers. Response times must be measured for the services modeled. Multiple samples of sufficient accuracy and precision are needed so that statistics can be computed (to determine variation in times).

We investigated a number of potentially promising approaches for capturing performance data in a portable way for MULE applications including profiling, logging, and custom interceptors. However, due to the demanding requirements for sufficient data, accuracy and precision we resorted to invasive instrumentation of the Loan Broker application using the java nanosecond timer (`java.lang.System.nanoTime()`, which provides nanosecond precision but not necessarily nanosecond accuracy), and logging (using the Apache logging framework, which has negligible performance overhead).

Using a single threaded JMeter load with 1,000 iterations we captured response times (in nanosecond precision) for the Loan Broker Application components. Table 1 shows the average measured times (in ms) for each component. Steps correspond to step numbers in Fig. 1. The Loan Broker component has two times, the initial getLoanQuote method (Step 2a), and the final ResponseAggregator method (Step 9a). The CreditAgencyGateway has four times corresponding to two Transformer components (Steps 4a and 4c), the "external" CreditAgency EJB component (Step 4b), and a MessageBuilder component (Step 4d). The Lender Service has times for the "external" Lender Service (Java Bean) (Step 6a), and a Lender Gateway Chaining Router component (Step 6b). The Banking Gateway time includes preparing and making each SOAP request (8a), and Step 8b is the time spent in the "external" Bank Web service.

The times parameterize the model as follows (shown in Fig. 2). The times spent by Mule in the LoanBroker service (2a, 9a) parameterize two calls to the MULE service from the LoanBroker composite service (1st and last steps). The three times measured for the Credit Agency Gateway (4a, 4c, 4d) are summed (8 ms = 2 + 4 + 2) and parameterizes a single call to the MULE service in the CreditAgencyGateway composite service (1st step). The BankingGateway time (8a) parameterizes a single

TABLE I. MEASURED PERFORMANCE TIMES USED FOR MODEL PARAMATERIZATION.

Step	Task	Time (ms)
2	2a: LoanBroker.getLoanQuote	1
4	4a: CreditAgencyGateWayService.Transformer	2
	4b: CreditAgency.EJB	1
	4c: CreditAgencyGateWayService.Transformer	4
	4d: CreditAgencyGateWayService.MessageBuilder	2
6	6a: LenderService.JavaBean	6
	6b: LenderGateway.ChainingRouter	4
8	8a: BankingGateway	10
	8b: BankService	22
9	9a: LoanBroker.ResponseAggregator	12

call to the MULE service from the BankingGateway composite service. Finally, the times for the "external" services (4b, 6a, 8b) parameterize the response times for the corresponding simple services.

In practice, there was significant variation in measured times. From our experience this is typical for Java applications as the Java Virtual Machine has a significant impact on performance in at least two ways: Garbage collection results in intermittent slower times, but adaptive HotSpot optimisation [7] tends to speed up times as an application runs. Time variation can be included in the model enabling accurate prediction of end-to-end response times and throughput.

V. MODELING SCENARIOS

We now show how the base model can be used and extended to explore increasingly realistic application and deployment scenarios, without having to re-measure the performance data.

A. Scenario 1: Single Server, single Bank Service.

In the first scenario we deployed the complete application, both MULE components and "external" services, on the same server (the default configuration). The Loan Broker ESB application was unmodified, and we used a loan request that obtained a quote from a single Bank web service only.

Using the initial performance model (with service response time parameters set as above, and server capacity set to two CPU cores) the simulation was run and the workload demand increased until the server saturated and the LoanBroker service response time dramatically increased. The predicted minimum end-to-end Loan Broker response time was 64ms, and the maximum capacity of the system was 32 transactions per second (TPS).

In order to validate the initial model so that alternative models could be explored with sufficient confidence (without further validation), we conducted load testing of the application using JMeter. Increasing the number of concurrent threads up to 40 we found that the average maximum capacity of the initial configuration was 38TPS. CPU usage on the server was close to 100% at this load. Increasing the load beyond 40 threads resulted in JMeter reporting HTTP errors. The model prediction of maximum capacity is therefore about 15% lower than measured, and is more pessimistic.

A number of factors may contribute to this, although some of them would tend to result in a more optimistic rather than pessimistic throughput prediction. We are not modeling or measuring the client/Loan Broker communication via HTTP/REST, but this overhead is included in the load tests. Average times for parameterization is not ideal, as median times are more "typical" as they exclude the bias resulting from outliers with very long times due to "abnormal" delays. The accuracy of the Java nanosecond timer needs to be further investigated. In particular, we noticed occasional negative times being reported (and excluded these times from the average). Negative values could possibly occur when tasks switch cores, due to drift in clock synchronization across multiple cores. Rounding the average times in ms introduces a small error, but overall it amounted to less than about 2%. Our modeling approach assumes that (unless modeled differently) times are CPU bound. For an infrastructure like MULE based on JMS messaging, and an application such as the ESB Loan Broker with a variety of different component types, it is likely that I/O and delays due to messaging queues may also contribute to the measured times.

The JMeter load test results did not show HTTP errors at 40 threads, but turning on a detailed result log revealed that some results were abnormal – i.e. the HTTP/REST call was succeeding, but there was some internal problem in the Loan Broker application. As exceptions result in part of the workflow/business logic not being executed, abnormal transactions would tend to result in a higher than expected capacity being measured. This theory is made more plausible by an observed increase in measured capacity above 40 threads when HTTP errors occurred in significant numbers.

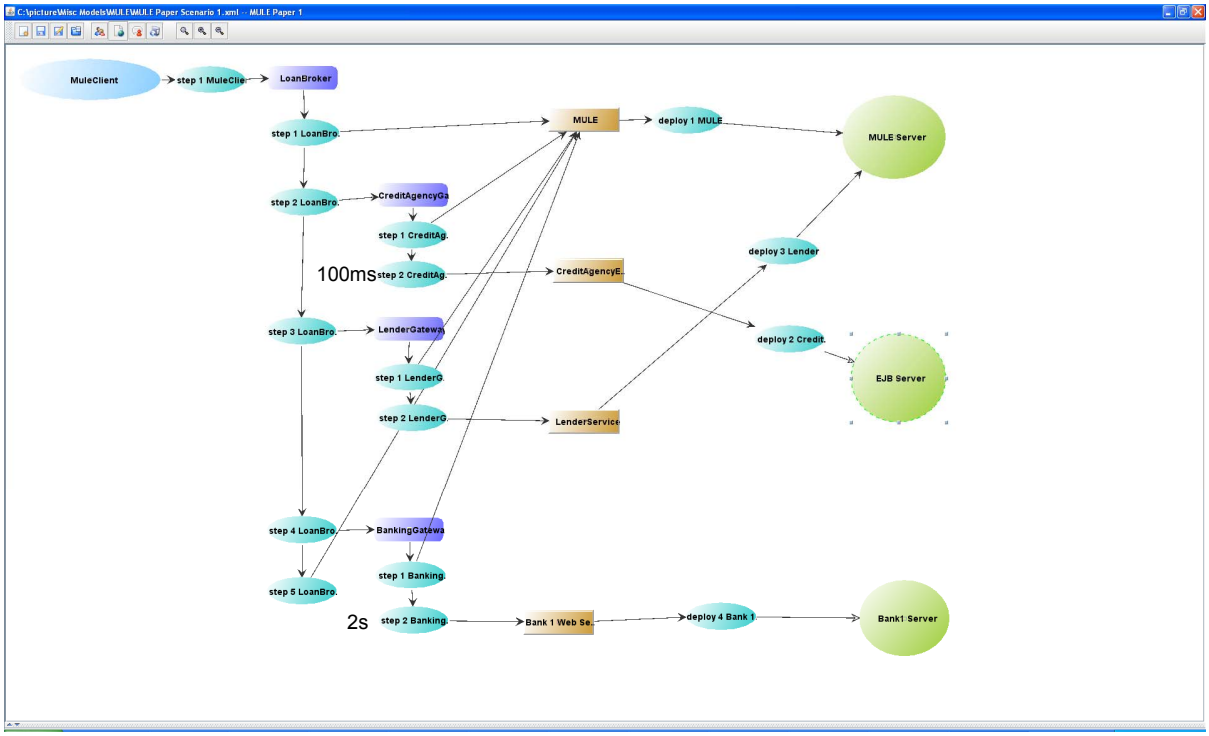

Figure 3. Screenshot of distributed deployment performance model (Scenarios 2 & 3)

B. *Scenario 2: Deployment of Credit Agency EJB and Bank Service on external servers.*

Scenario 2 explores a more realistic deployment of a distributed Loan Broker application. In this scenario we deploy the Credit Agency EJB and the Bank Service to external servers (two circles on the lower right in Fig. 3) with sufficient CPUs to ensure that they are not a bottleneck. The only times spent on the MULE server are now from the MULE LoanBroker service and gateway components, and the Lender Service JavaBean. Service times are assumed to be the same as for the initial centralised scenario.

The model now predicts a maximum throughput (with no increase in response time) of 47TPS (c.f. 32 TPS for centralised deployment, an increase of 15TPS). This increase in throughput is to be expected as hosting some of the services externally means that the MULE server has spare capacity which can be utilized by the gateway services.

C. *Scenario 3: Realistic external service times.*

The external services in the Loan Broker application actually do very little work, resulting in shorter than realistic services times. For this scenario we increase the average external service times to more realistic times as follows: CreditAgency EJB = 100ms, Bank1 = 2s, Bank2 = 2.5s, Bank3 = 3s, Bank4 = 3.5s (Bank2-Bank4 are used in Scenario 4 below). The increased times for the EJB and Bank1 are shown in Fig. 3. The predicted Loan Broker service response time is 2.141s, and the throughput is unchanged (47TPS), as the demand on the MULE server hasn't changed from scenario 2, and we assume sufficient capacity of EJB and Bank servers to cope with demand with no increase in response time. We also assume that service response times include round-trip network latency. In practice for EJBs latency will be close to zero, but latency may be several seconds for internet based web services. Network latency can easily be modeled explicitly, but for simplicity has been excluded here.

D. *Scenario 4: Four Bank Web Services.*

The Loan Broker application uses up to four Banking services to obtain a quote. The Bank services are selected by the Lender service based on requested loan parameters, Credit History, and knowledge of the Banks lending preferences. Examining the code and run-time traces we found that there is a constant overhead per Bank service call (due to preparation and invocation of each bank service via SOAP), and that the Loan Broker Response Aggregator time increases with the number of Bank services. We changed the model to reflect this, and also modeled the probability of multiple Bank services being called (concurrently) as follows: Probability 0.2 of 1 service (Bank1 only); Probability 0.5 of 2 services (Bank2 and Bank3); Probability 0.3 of 3 services (Bank2-Bank4). Once invoked, the bank services and replies to the Loan Broker are processed concurrently.

The modified model predicts that the maximum capacity is reduced by 17TPS from 47TPS to 30 TPS, and the response time varies from a minimum of 2.141s to a maximum of 4.4s. The tool also computes statistical metrics such as response time distributions over extended simulation run times. For 2,000 Loan Broker service requests the distribution of response times was as follows: 2.1s (minimum), 3.3s (50%), 4.1s (99%), 4.4s (100%, maximum).

E. Scenario 5: Bank Service Level Agreements (SLAs).

Scenario 4 was based on Bank services taking average times only. However, in practice external web services exhibit variations in times which may be described in a service level agreement (SLA). Assume Bank services have SLAs which promise the average times above in 99% of invocations, but with 1% chance of taking double the average time. What impact does this have on the SLA the Loan Broker service can offer to consumers? The predicted distribution of response times for the Loan Broker service over 2,000 requests ranged from 2.141s to 7.8s, with 99% below 6.1s. The maximum increased by 3.4s from 4.4s to 7.8s, and the 99% value has increased by 2s from 4.1s to 6.1s.

F. Scenario 6: Service metrics – Timeouts and Threads.

The SOPM approach predicts service metrics which are critical for understanding SOA performance. These are used for developing SLAs between service consumers and providers (both internal and external), and for ensuring adequate resources for services under different loads (e.g. concurrency in services or servers can be used to model thread or connection pool sizes, or total memory requirements).

MULE allows service components to have default timeouts to enable exceptions to be detected and actions taken (e.g. compensating transactions). Models of MULE applications can be used to predict maximum timeouts under normal operation to assist with configuring realistic service timeout values. For example, the MULE Loan Broker application has a default timeout of 10s for the Loan Broker service. Transient peak demands may result in higher than expected response times due to occasional saturation of MULE resources. The simulation predicted a maximum Loan Broker service time in excess of 10s (10.5s), given a Poisson arrival distribution (with unchanged arrival rate for an average throughput of 30TPS as in the previous scenario), indicating that the Loan Broker timeout must be increased to prevent premature timeouts.

Tuning MULE is currently a manual and complex time consuming manual process [8], as the number of threads and thread management policy can be specified for each service receiver, component, and dispatcher. A SOPM can assist with determining the number of threads required as it predicts metrics for each service, including the number of concurrent users (which is a proxy for the number of threads required). For example, the model predicts that up to 80 threads may be needed for the CreditAgencyGateway component.

VI. Related Work

NICTA's Service-Oriented Performance Modeling technology was developed to address the business needs of real SOA projects, initially in Australian e-Government. Preliminary approaches were reported in [12, 3]. However, we found that there were problems applying these pioneering approaches to larger SOAs. We explored other reported approaches and tools for SOA performance analysis and modeling [10, 11, 13, 14] but found they had similar problems. In particular: they do not scale well to larger or more complex systems (e.g. with complex composite services, large numbers of services and servers, complex orchestration or workloads, server virtualization, etc); they do not answer all the performance questions needed (e.g. queue network solutions do not predict service specific metrics for multiple services deployed on a shared server, and analytical solutions cannot model saturated or dynamic behaviour of a SOA, such as time to recover from overload); and they are too complex to build, use, and maintain by software architects.

We developed our SOPM method and model-driven tool support to enable rapid construction of service-oriented performance models to answer service-specific performance questions [2]. Subsequently we refined and validated the applicability, cost, and business value of this approach on other real projects with different architectures. Other work on ESB scalability and performance is reported in [1, 6]. Our approach corroborates the results of [1], but it was significantly easier to develop and modify models and predict service-specific performance metrics and statistics.

VII. Conclusions and Future work

This paper demonstrates that performance models of ESBs can successfully be built and used with our SOPM approach. Models were built with direct correspondence between ESB/application components and model components, making it easy to construct, parameterize, use, modify, and maintain them. Measurements of component response times were obtained from the running application on the test-bed configuration, and the times were mapped directly to model component parameters. The capacity predictions of the model for scenario 1 were within 15% of measured throughput, which gives reasonable confidence in the accuracy of the model to extend it to cases where validation may not be possible. Metrics for services such as response times (to assist with setting service timeouts), concurrency (to assist with setting thread limits), and throughput (to assist with negotiating SLAs with providers and consumers of services) were able to be predicted for the models built. Finally, more realistic deployments scenarios were modeled and performance predictions made.

Future work is planned to:

- Investigate better ways of obtaining improved quality and quantity of performance measurements from running ESB infrastructure and applications, to increase the ease and accuracy of model parameterization. This will enable modeling of time variation within the MULE services and produce more accurate models of variability in application times.

- Refine the modeling and load testing approaches for ESBs to understand the causes and reduce the discrepancy between the predicted and measured results.
- Model and validate the alternative examples of the MULE Loan Broker application in order to predict variations in applications. In practice, multiple distributed ESBs are used together in various topologies, and host more than one application. We plan to explore the modeling and predictive accuracy of the models for multiple ESBs and applications.
- Extend and test the approach with different ESB technologies and products.

ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. Thanks to our NICTA/ANU summer scholar Gyurme Dahdul (Auckland University, New Zealand) for assistance with the MULE experiments.

REFERENCES

[1] Yan Liu, Ian Gorton, Liming Zhu, Performance Prediction of Service-Oriented Applications based on an Enterprise Service Bus. Computer Software and Applications Conference 2007, (COMPSAC 2007), Beijing, Volume 1, 327-334.

[2] Brebner, P. C. Performance modeling for service oriented architectures. In *Companion of the 30th international Conference on Software Engineering*. ICSE Companion '08. ACM, New York, NY, 953-954.

[3] Paul Brebner, Liam O'Brien, Jon Gray. Performance Modeling for e-Government Service Oriented Architectures (SOAs). 19th Australian Software Engineering Conference. ASWEC 2008. pp. 130-138.

[4] O'Brien, L., Brebner, P., and Gray, J. 2008. Business transformation to SOA: aspects of the migration and performance and QoS issues. In *Proceedings of the 2nd international Workshop on Systems Development in SOA Environments*. SDSOA '08. 35-40.

[5] MULE ESB, http://www.mulesource.com/

[6] Ken Ueno, Michiaki Tatsubori, Early Capacity Testing of an Enterprise Service Bus. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006),* 709-716, September, 2006.

[7] Paleczny, M., Vick, C., and Click, C. 2001. The java hotspot™ server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, USENIX.

[8] Jackie Wheeler, Performance Tuning in Mule, November 25, 2008, http://blog.mulesource.org/2008/11/performance-tuning/

[9] Gregor Hohpe, Bobby Woolf, Enterprise Integration Patterns Book, Loan Broker application, http://www.eaipatterns.com/ComposedMessagingWS.html

[10] Marin Litoiu, *Migrating to web services: a performance engineering approach*, Journal of Software Maintenance and Evolution: Research and Practice, Volume 16, Issue, 51-70, 2004.

[11] Grundy, J., Hosking, J., Li, L., and Liu, N. 2006. Performance engineering of service compositions. In *Proceedings of the 2006 international Workshop on Service-Oriented Software Engineering*. SOSE '06. ACM, New York, NY, 26-32.

[12] Yan Liu, Liming Zhu, Ian Gorton, Performance Assessment for e-Government Services: An Experience Report, Component-Based Software Engineering (CBSE) 2007, LNCS, Springer, Volume 4608/2007, 74-89.

[13] Sloane, E. Way, T. Gehlot, V. Beck, R. Solderitch, J. Dziembowski, E. A Hybrid Approach to Modeling SOA Systems of Systems Using CPN and MESA/Extend, 1st Annual IEEE Systems Conference, 2007, April 2007, 1-7.

[14] Bertoli, M. Casale, G. Serazzri, G., Java Modelling Tools: an Open Source Suite for Queueing Network Modelling and Workload Analysis, 3rd International Conference on the Quantitative Evaluation of Systems (QEST 2006), 2006, Riverside, CA. 119-120.

[15] MULE Loan Broker Example, http://www.mulesource.org/display/MULE2INTRO/Loan+Broker+Example

[16] MULE ESB Loan Broker Application, http://www.mulesource.org/display/MULE2INTRO/Loan+Broker+ESB+Example

[17] Sharp, J. 2007 Microsoft® Windows® Communication Foundation Step by Step. Microsoft Press.