Early Performance Testing of Distributed Software Applications*

Giovanni Denaro†
Universitá di Milano Bicocca
Dipartimento di Informatica,
Sistemistica e Comunicazione
I-20126, Milano, Italy
denaro@disco.unimib.it

Andrea Polini†
ISTI-CNR
Via Moruzzi, 1
I-56124 Pisa, Italy
a.polini@sssup.it

Wolfgang Emmerich
University College London
Dept. of Computer Science
WC1E 6BT, London, UK
w.emmerich@cs.ucl.ac.uk

ABSTRACT

Performance characteristics, such as response time, throughput and scalability, are key quality attributes of distributed applications. Current practice, however, rarely applies systematic techniques to evaluate performance characteristics. We argue that evaluation of performance is particularly crucial in early development stages, when important architectural choices are made. At first glance, this contradicts the use of testing techniques, which are usually applied towards the end of a project. In this paper, we assume that many distributed systems are built with middleware technologies, such as the Java 2 Enterprise Edition (J2EE) or the Common Object Request Broker Architecture (CORBA). These provide services and facilities whose implementations are available when architectures are defined. We also note that it is the middleware functionality, such as transaction and persistence services, remote communication primitives and threading policy primitives, that dominate distributed system performance. Drawing on these observations, this paper presents a novel approach to performance testing of distributed applications. We propose to derive application-specific test cases from architecture designs so that performance of a distributed application can be tested using the middleware software at early stages of a development process. We report empirical results that support the viability of the approach.

Keywords

Middleware, Software Testing, Distributed Software Architecture, Software Performance, Performance Analysis Models

†The research was conducted while this author was on study leave at UCL.

*This work has been partially supported by the European Union through the Research Training Network *SegraVis*.

1. INTRODUCTION

Various commercial trends have lead to an increasing demand for distributed applications. Firstly, the number of mergers between companies is increasing. The different divisions of a newly merged company have to deliver unified services to their customers and this usually demands an integration of their IT systems. The time available for delivery of such an integration is often so short that building a new system is not an option and therefore existing system components have to be integrated into a distributed system that appears as an integrating computing facility. Secondly, the time available for providing new services are decreasing. Often this can only be achieved if components are procured off-the-shelf and then integrated into a system rather than built from scratch. Components to be integrated may have incompatible requirements for their hardware and operating system platforms; they have to be deployed on different hosts, forcing the resulting system to be distributed. Finally, the Internet provides new opportunities to offer products and services to a vast number of potential customers. The required scalability of e-commerce or e-government sites cannot usually be achieved by centralised or client-server architectures but demand the use of distributed software architectures.

Given the growing importance of distributed systems, we are interested in devising systematic ways to ascertain that a given distributed software architecture meets performance requirements of their users. Performance can be characterised in several different ways. *Latency* typically describes the delay between request and completion of an operation. *Throughput* denotes the number of operations that can be completed in a given period of time. *Scalability* identifies the dependency between the number of distributed system resources that can be used by a distributed application (typically number of hosts or processors) and latency or throughput. Despite the practical significance of these various aspects it is still not adequately understood how to test the performance of distributed applications.

Weyuker and Vokolos reported on the weakness of the published scientific literature on *software performance testing* in [26]. To this date no significant scientific advances have been made on performance testing. Furthermore the set of tools available for software performance testing is fairly limited. The most widely used tools are workload generators

and performance profilers that provide support for test execution and debugging, but they do not solve many unclear aspects of the process of performance testing. In particular, researchers and practitioners agree that the most critical performance problems depend on decisions made in the very early stages of the development life cycle, such as architectural choices. Even though iterative and incremental development has been widely promoted [18, 10, 12], the testing techniques developed so far are very much focused on the end of the development process.

As a consequence of the need for early evaluation of software performance and the weakness of testing, the great majority of research has focused on performance analysis models [1, 20, 19, 2, 3, 23] rather than testing techniques. This research shares in general the approach of translating architecture designs, mostly given in the Unified Modeling Language (UML [5]), to models suitable for analysing performance, such as, Layered Queuing Networks (e.g. [19]), Stochastic Petri Nets (e.g. [2]) or stochastic process algebras (e.g. [20]). Estimates of performance are used to reveal flaws in the original architecture or to compare different architectures and architectural choices. Models may give useful hints of the performance and help identify bottlenecks, however they tend to be rather inaccurate. Firstly, models generally ignore important details of the deployment environment. For example, performance differences may be significant when different databases or operating systems are used, but the complex characteristics of specific databases and operating systems are very seldom included in the models. Secondly, models often have to be tuned manually. For example, in the case of Layered Queued Networks, solving contention of CPU(s) requires, as input, the number of CPU cycles that each operation is expected to use. Tuning of this type of parameters is usually guessed by experience and as a result it is not easy to obtain precise models.

With the recent advances in distributed component technologies, such as J2EE [22] and CORBA [17], distributed systems are no longer built from scratch [7]. Modern distributed applications often integrate both off-the-shelf and legacy components, use services provided by third-parties, such as real-time market data provided by Bloomberg or Reuters, and rely on commercial databases to manage persistent data. Moreover, they are built on top of middleware products (hereafter referred to as *middlewares*), i.e., middle-tier software that provides facilities and services to simplify distributed assembly of components, e.g., communication, synchronisation, threading and load balancing facilities and transaction and security management services [8]. As a result of this trend, we have a class of distributed applications for which a considerable part of their implementation is already available when the architecture is defined (e.g. during the Elaboration phase of the Unified Process). In this paper, we argue that this enables performance testing to be successfully applied at these early stages.

The main contribution of this paper is the description and evaluation of a method for testing performance of distributed software in the early development stages. The method is based on the observation that the middleware used to build a distributed application often determines the overall performance of the application. For example, middlewares and databases usually contain the software for transaction and persistence management, remote communication primitives and threading policies, which have great impact on the different aspects of performance of distributed systems. However, we note that only the coupling between the middleware and the application architecture determines the actual performance. The same middleware may perform very differently in the context of different applications. Based on these observations, we propose using architecture designs to derive application-specific performance test cases that can be executed on the early available middleware platform a distributed application is built with. We argue that this allows empirical measurements of performance to be successfully done in the very early stages of the development process. Furthermore, we envision an interesting set of practical applications of this approach, that is: evaluation and selection of middleware for specific applications; evaluation and selection of off-the-shelf components; empirical evaluation and comparison of possible architectural choices; early configuration of applications; evaluation of the impact of new components on the evolution of existing applications.

The paper is further structured as follows. Section 2 discusses related work and highlights the original aspects of our research. Section 3 gives details of our approach to performance testing. Section 4 reports about the results of an empirical evaluation of the main hypothesis of our research, i.e., that the performance of distributed application can be successfully measured based on the early-available components. Section 5 discusses the limitations of our approach and sketches a possible integration with performance modelling techniques. Finally, Section 6 summarises the contributions of the paper and sketches our future research agenda.

2. RELATED WORK

In this section, we briefly review related work in the areas of performance testing of distributed applications and studies on the relationships between software architecture and middleware.

Performance testing of distributed applications

Some authors exploited empirical testing for studying the performance of middleware products. Gorton and Liu compare the performance of six different J2EE-based middlewares [9]. They use a benchmark application that stresses the middleware infrastructure, the transaction and directory services and the load balancing mechanisms. The comparison is based on the empirical measurement of throughput per increasing number of clients. Similarly, Avritzer et al. compare the performance of different ORB (Object Request Broker) implementations that adhere to the CORBA Component Model [13]. Liu and al. investigate the suitability of micro-benchmarks, i.e., light-weight test cases focused on specific facilities of the middlewares, such as, directory service, transaction management and persistence and security support [14].

This work suggests the suitability of empirical measurement for middleware selection, i.e, for making decisions on which middleware will best satisfy the performance requirements of a distributed application. However, as Liu et al. remark in the conclusions of their paper ([14]), "how to incorpo-

rate application-specific behaviour in the equations and how far the results can be generalised across different hardware platforms, databases and operating systems, are still open problems." Our research tackles these problems. We study application-specific test cases for early performance evaluation (or also comparing) the performance of distributed applications in specific deployment environments, which include middlewares, databases, operating systems and other off-the-shelf components.

Weyuker and Vokolos report on the industrial experience of testing the performance of a distributed telecommunication application at AT&T [26]. They stress that, given the lack of historical data on the usage of the target system, the architecture is key to identify software processes and input parameters (and realistic representative values) that will most significantly influence the performance.

Our work extends this consideration to a wider set of distributed applications, i.e., distributed component-based software in general. Moreover, we aim to provide a systematic approach to test-definition, implementation and deployment that are not covered in the work of Weyuker and Vokolos.

Software architecture and middleware

Medvidovic, Dashofy and Taylor state the idea of coupling the modelling power of software architectures with the implementation support provided by middleware [15]. They notice that "architectures and middleware address similar problems, that is large-scale component-based development, but at different stages of the development life cycle." They propose to investigate the possibility of defining systematic mappings between architectures and middlewares. To this end, they study the suitability of a particular element of software architecture, the *software connector*. Metha, Phadke and Medvidovic himself moreover present an interesting taxonomy of software connectors [16].

Although they draw on similar assumptions (i.e., the relationships between architecture and middleware), our research and that of Medvidovic et al. have different goals: We aim at measuring performance attributes of an architecture based on the early available implementation support (which the middleware is a significant part of); Medvidovic and colleagues aim at building implementation topologies (e.g., bridging of middlewares) that preserve the properties of the original architecture. However, the results of previous studies on software connectors and the possibility of mapping architectures on middlewares may be important for engineering our approach.

3. EARLY TESTING OF PERFORMANCE

In this section, we introduce our approach to early performance testing of distributed component-based applications. We also focus on the aspects of the problem that need further investigation. Our long-term goal is to realize a software tool that supports the application of the approach we describe below.

Our approach comprises a performance testing process that consists of the following phases:

1. Selection of the use-case scenarios (hereafter referred to simply as *use-cases*) relevant to performance, given a set of architecture designs.
2. Mapping of the selected use-cases to the actual deployment technology and platform.
3. Generation of *stubs* of components that are not available in the early stages of the development life cycle, but are needed to implement the use cases.
4. Execution of the test, which in turn includes: deployment of the Application Under Test (AUT), creation of workload generators, initialisation of the persistent data and reporting of performance measurements.

We now discuss the research problems and our approach to solving them for each of the above phases of the testing process.

Selecting performance use-cases

The design of test suites for performance testing is radically different from the case of functional testing. (This has been noticed by many authors, e.g. [26].) In performance testing, the functional details of the test cases, i.e., the actual values of the inputs, are generally of little importance. Table 1 classifies the main parameters relevant to performance testing of distributed applications. First, important concerns are traditionally associated with workloads and physical resources, e.g., the number of users, the frequencies of inputs, the duration of the test, the characteristics of the disks, the network bandwidth and the number and speed of CPU(s). Next, it is important to consider the middleware configuration, for which the table reports parameters in the case of J2EE-based middlewares. Here, we do not comment further on workload, physical resource and middleware parameters, which are extensively discussed in the literature (e.g., [26, 25, 14]).

Other important parameters of performance testing in distributed settings can be associated with interactions among distributed components and resources. Different ways of using facilities, services and resources of middlewares and deployment environments likely correspond to different performance results. Performance will differ if the database is accessed many times or never. A given middleware may perform adequately for applications that stress persistence and quite badly for transactions. In some cases, middlewares may perform well or badly for different usage patterns of the same service. Application-specific cases for performance should be given such that the most relevant interactions specifically triggered by the AUT are covered.

The last row of Table 1 classifies the relevant interactions in distributed settings according to whether they take place between the middleware and the components, among the components themselves[1] or to access persistent data in a database. This taxonomy is far from complete and we are

[1]Although interactions among distributed components map on interactions that take actually place at the middleware level, they are elicited at a different abstraction level and thus they are considered as a different category in our classification.

Workload	Number of clients Client request frequency Client request arrival rate Duration of the test
Physical resources	Number and speed of CPU(s) Speed of disks Network bandwidth
Middleware configuration	Thread pool size Database connection pool size Application component cache size JVM heap size Message queue buffer size Message queue persistence
Application specific	Interactions with the middleware - use of transaction management - use of the security service - component replication - component migration Interactions among components - remote method calls - asynchronous message deliveries Interactions with persistent data - database accesses

Table 1: Performance parameters

conducting further studies and empirical assessments in this direction. However, we believe that a taxonomy of distributed interactions is key for using our approach. Based on such a taxonomy, our next step is the definition of appropriate metrics to evaluate the performance relevance of the available use-cases according to the interactions that they trigger. This not only supports the selection of sets of use-cases so that they adequately represent the performance of the target application but also enables the definition of a reasonable distribution of the selected use-cases in the final workload, according to the basic principles of synthetic workload modelling [11, 24].

Mapping use-cases to middleware

In the initial stages of a software process, software architectures are generally defined at a very abstract level. Often, they just describe the business logic and abstract many details of deployment platforms and technologies. One of the expected strengths of our approach is indeed the possibility of driving software engineers through the intricate web of architectural choices, technologies and deployment options, keeping the focus on the performance of the final product. To this end, however, we need to understand how abstract use-cases are mapped to possible deployment technologies and platforms.

Let us for example consider the use-case fragment in Figure 1(a), which represents in UML the creation of a component C of type `Component` issued by a client with the message `create`. The deployment of such use-case on a J2EE-based middleware will likely map on a component (namely, an Enterprise Java Bean) with the interactions shown in Figure 1(b). The message `create` from the client will be actually delivered to a component container that is part of the middleware. The middleware will then dispatch the request to the actual component C with two messages, `create` and `post_create`, sent respectively before and after the initialisation of the support structures for C within the middleware. Optionally, the client will use the directory service provided by the middleware, through the Java Naming and Directory Interface (JNDI), to retrieve the component factory responsible for creating C. Furthermore, we ought to specify the transactional behaviour, e.g., whether the method will execute in the context of the client transaction or a newly created (nested) transaction.

To address the mapping of abstract use-cases to specific deployment technologies and platforms, we intend to leverage recent studies on software connectors [16, 15]. Well characterised software connectors may be associated with deployment topologies that preserve the properties of the original architecture. Also, possible implementation alternatives can be defined in advance as characteristics of the software connectors. For instance, with reference to example above, we would like to define a component creation connector that provides properties for the case showed in Figure 1(a). Then, we would like to associate with this connector, the J2EE deployment topology given in Figure 1(b) and the list of the possible transaction behaviours. Deployment topologies and alternatives might vary for different deployment technologies and platforms. Previous studies on software connectors are precious references in this direction, but further work is still needed to understand the many dimensions and species of software connectors and their relationships with the possible deployment technologies and platforms.

Generating stubs

So far, we have suggested that early test cases of performance can be derived from instances of architecture use-cases and that software connectors can be exploited as a means to establish the correspondence between the abstract views provided by the use-cases and the concrete instances. However, to actually implement the test cases, we must also solve the problem that not all the application components that participate in the use-cases are available in the early stages of the development life cycle. For example, the components that implement the business logic are seldom available, although they participate in most of the use-cases. Our approach uses *stubs* in place of the missing components.

Stubs are fake versions of components that can be used instead of the corresponding components for instantiating the abstract use-cases. In our approach, stubs are specifically adjusted to use-cases, i.e., different use-cases will require different stubs of the same component. Stubs will only take care that the distributed interactions happen as specified and the other components are coherently exercised. Our idea of the engineered approach is that the needed stubs are automatically generated based on the information contained in use-cases elaborations and software connectors. For example, referring once again to Figure 1, if the component C was not available, its stub would be implemented such that it is just able to receive the messages `create` through the actual middleware. It would contain empty code for the methods `create` and `post_create`, but set the corresponding transaction behaviour as specified. In a different case, if the persistence of the instances of C and the corresponding interactions with the database, were specified in the use-case,

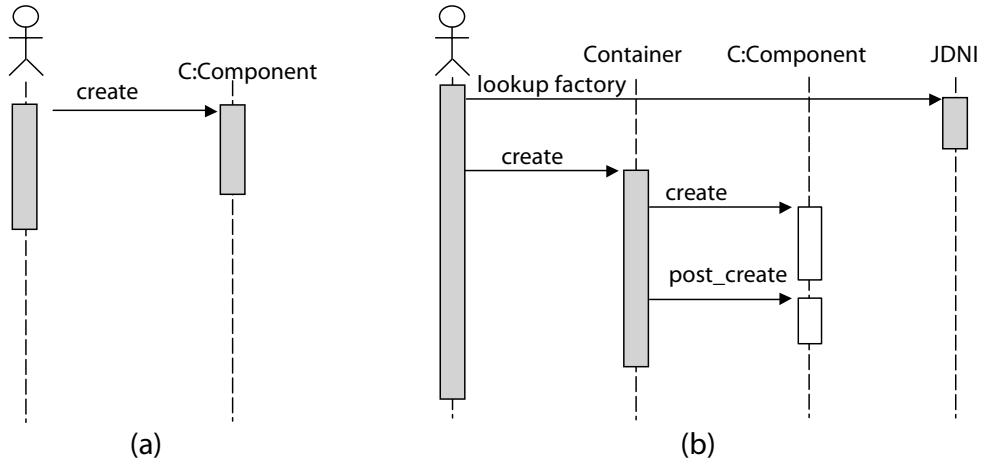

Figure 1: An abstract use-case (a) and its deployment view (b)

the database calls (as far as they can be extracted from the use-case) would be included in the method `create` of C. If it was also the case that the database was an early-available component, the actual SQL code for the stated interactions would be hard-coded in the stub of C. Of course, many functional details of C are generally not known and cannot be implemented in the stub. Normally, this will result in discrepancies between execution times within the stubs and the actual components that they simulate.

The main hypothesis of our work is that performance measurements in the presence of the stubs are good enough approximations of the actual performance of the final application. This descends from the observation that the available components, e.g., middlewares and databases, embed the software that mainly impact performance. The coupling between such implementation support and the application-specific behaviour can be extracted from the use-cases, while the implementation details of the business components remain negligible. In other words, we expect that the discrepancies of execution times within the stubs are orders of magnitude less than the impact of the interactions facilitated by middleware and persistence technology, such as databases. We report a first empirical assessment of this hypothesis in Section 4 of this paper, but are aware that further empirical studies are needed.

Executing the test

Building the support to test execution shall mostly involve technical rather than scientific problems, at least once the research questions stated above have been answered. Part of the work consists of engineering the activities of selecting the use-cases, mapping them to deployment technologies and platforms, and generating the stubs to replace missing components. Also, we must automate deployment and implementation of workload generators, initialisation of persistent data, execution of measurements and reporting of results. No relevant aspects for research are directly connected with this.

4. PRELIMINARY ASSESSMENT

This section empirically evaluates the core hypothesis of our research, i.e., that the performance of a distributed application can be successfully tested based on the middleware and/or off-the-shelf components that are available in the early stages of the software process. To this end, we conducted an experiment in a controlled environment. First, we considered a sample distributed application for which we had the whole implementation available. Then, we selected an abstract use-case of the application and implemented it as a test case based on the approach described in Section 3. Finally, we executed the performance test (with different amounts of application clients) on the early available components and compared the results with the performance measured on the actual application.

Experiment setting

As for the target application, we considered the *Duke's Bank application* presented in the J2EE tutorial [4]. This application is distributed by Sun under a public license, thus we were able to obtain the full implementation easily. The Duke's bank application consists of 6,000 lines of Java code that is meant to exemplify all the main features of the J2EE platform, including the use of transactions and security. We consider the Duke's bank application to be adequately representative of medium-size component-based distributed applications. The Duke's bank application is referred to as DBApp in the remainder of this paper.

The organisation of the DBApp is given in Figure 2 (borrowed from [4]). The application can be accessed by both Web and application clients. It consists of six EJB (Enterprise Java Beans [22]) components that handle operations issued by the users of a hypothetic bank. The six components can be associated with classes of operations that are related to bank accounts, customers and transactions, respectively. For each of these classes of operations a pair of session bean and entity bean is provided. Session beans are responsible for the interface towards the users and the entity beans handle the mapping of stateful components to underlying database table. The arrows represent the possi-

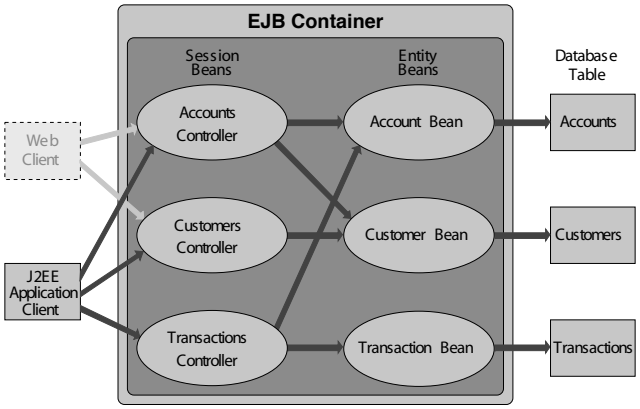

Figure 2: The Duke's Bank application

ble interaction patterns among the components. The EJBs that constitute the business components are deployed in a single container within the application server (which is part of the middleware). For the experiment we used the JBoss application server and the MySql database engine, running on the same machine.

Then, we selected a sample use-case that describes the transfer of funds between two bank accounts. Figure 3 illustrates the selected use-case in UML. A client application uses the service `Transfer` provided by the DBApp. This service requires three input parameters, representing the two accounts and the amount of money, respectively involved in the transfer. The business components of the DBApp realize the service using the database for storing the persistent data: the database is invoked four times, for updating the balances of the two accounts and recording the details of the corresponding transactions. We assume that the database engine is software that is available early in the software process. Thus, for the test, we used the same database engine, table structure and SQL code than in the original application. This is why we represented the database as a shadowed box in the figure. Differently from the database, the business components of the DBApp are assumed to be not available, thus we had to generate corresponding stubs.

For implementing the stubs, we had to map the abstract use-

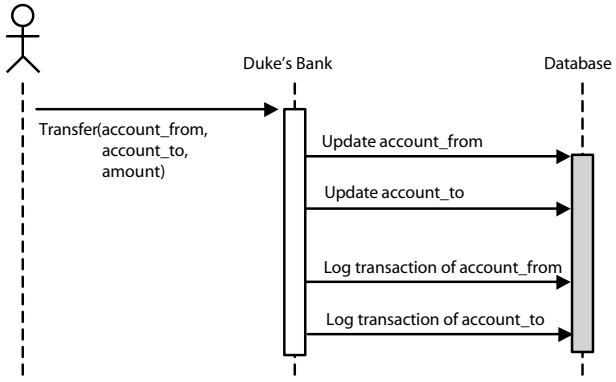

Figure 3: A sample use-case for the Duke's Bank

case on the selected deployment technology, namely J2EE. We already commented on the role that software connectors may play in the mapping. However, our research is not yet mature enough to exploit software connectors for this purpose. Presently, we just manually augmented the use-case with the necessary information as follows. As for the interaction between the clients and the DBApp, we specified that the service `Transfer` is invoked as a synchronous call and starts a new transaction. As for the interaction between the DBApp and the database, we specified that: the four invocations are synchronous calls that take place in the context of a single transaction and embed the available SQL code; the database connection is initialised for each call[2]; the DBApp uses entity beans and bean managed persistence[3] to handle the interactions with the database tables. Based on this information, we implemented the stubs as needed to realize the interactions in the considered use-case and we deployed the test version of the DBApp (referred to as DBTest) on the JBoss application server.

Finally, we implemented a workload generator and initialised the persistent data in the database. The workload generator is able to activate a number of clients at the same time and takes care of measuring the average response time. For the persistent data, we instantiated the case in which each client withdraws money from its own account (i.e., there exists a bank account for each client) and deposits the corresponding amount to the account of a third party, which is supposed to be the same for all clients. This simulates the recurrent case in which a group of people is paying the same authority over the Internet. Incidentally, we notice that, in an automated test environment, initialisation of persistent data would only require to specify the performance sensible part of the information, while the actual values in the database tables are generally of little importance. For example, in our case, only the number of elements in each table and the relationships with the instanced use-case, i.e., whether all clients access the same or a different table row, are the real concerns.

With reference to the performance parameters of Table 1, we generated a workload, to test both DBApp and DBTest, with increasing numbers of clients starting from one to one hundred. The two applications were deployed on a JBoss 3.0 application server running on a PC equipped with a Pentium III CPU at 1 GHz, 512 MB of RAM memory and the Linux operating system. To generate the workload we run the clients on a Sun Fire 880 equipped with 4 Sparc CPUs at 850 MHz and 8 GB of RAM. These two machines were connected via a private local area network with a bandwidth of 100 MBit/sec. For the stubs we used the same geographical distances as the components of the actual application. Moreover, in order to avoid influences among the experiments that could be caused by the contemporary existence of a lot of active session beans, we restarted the application

[2]Although this may sound as a bad implementation choice, we preferred to maintain the policy of the original application to avoid biases on the comparison.

[3]Entity beans are J2EE components that represent persistent data within an application. Each database table is associated to an entity bean. The data in the entity bean are taken synchronised with the database. In the case of bean managed persistence the synchronisation code is embedded in the entity bean.

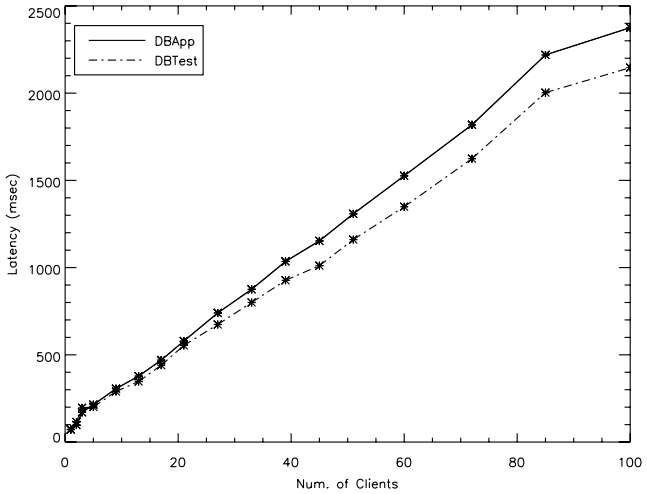

Figure 4: Latency of DBApp and DBTest for increasing numbers of clients

server between two successive experiments. JBoss has been used running the default configuration. Finally, the specific setting concerning the particular use case, as already discussed in the previous paragraphs, foresaw the use of remote method calls between the components and the use of the transaction management service, in order to handle the data shared by the various beans consistently.

Experiment results

We have executed both DBApp and DBTest for increasing numbers of clients and measured the latency for the test case. We repeated each single experiment 15 times and measured the average latency time. Figure 4 shows the results of the experiments. It plots the latency time of both DBApp and DBTest against the number of clients, for all the repetitions of the experiment. We can see that the two curves are very near to each other. The average difference accounts for the 9.3% of the response time. The experiments also showed a low value for the standard deviation. The ratio between σ and the expectation results, in fact, definitively lower of the 0.15, both for the DBApp and for the DBTest.

The results of this experiment suggest the viability of our research because they witness that the performance of the DBApp in a specific use-case is well approximated by the DBTest, which is made out of the early-available components. However, although the first results are encouraging, we are aware that a single experiment cannot be generalised. We are now working on other experiments to cover the large set of alternatives of component-based distributed applications. We plan to experiment with different use-cases, sets of use-cases for the same test case, different management schemas for transactions and performance, different communication mechanisms such as asynchronous call, J2EE-based application server other than JBoss, CORBA-based middlewares, other commercial databases and in the presence of other early-available components.

5. SCOPE AND EXTENSIONS

Our results support the possibility that using stubs for the application code, but the real middleware and database proposed for the application, can provide useful information on the performance of a distributed application. This is particularly true for enterprise information system applications that are based on distributed component technologies, such as J2EE and CORBA. We have already commented that for this class of distributed applications the middleware is generally responsible for most of the implementation support relevant to performance, e.g., mechanisms for handling distributed communication, synchronisation, persistence of data, transactions, load balancing and threading policies. Thus in most cases critical contention of resources and bottlenecks happen at the middleware level, while the execution time of the business components is negligible.

Our approach allows providers of this class of distributed applications to test whether, and to which extent, a given middleware may satisfy the performance requirements of an application that is under development. In this respect, our approach may perform better than pure benchmarking of middlewares (e.g., [9, 13, 14]), because it enables application-specific evaluation, i.e., it generates test cases that take into account the specific needs of a particular business logic and application architectures. Moreover, the approach has a wider scope than solely testing the middleware. It can be generalised to test all components that are available at the beginning of the development process, for example, components acquired off-the-shelf by third parties. Based on the empirical measurements of performance, tuning of architectures and architectural choices may also be performed.

Despite these valuable benefits, however, we note that our approach cannot identify performance problems that are due to the specific implementation of late-available components. For example, if the final application is going to have a bottleneck in a business component that is under development, our approach has no chance to discover the bottleneck that would not be exhibited by a stub of the component. Performance analysis models remain the primary reference to pursue evaluation of performance in such cases.

Currently, we are studying the possibility of combining empirical testing and performance modelling, aiming at increasing the relative strengths of each approach. In the rest of this section we sketch the basic idea of this integration.

One of the problem of applying performance analysis to middleware-based distributed systems is that the middleware is in general very difficult to represent in the analysis models. For instance, let us consider the case in which one wants to provide a detailed performance analysis of the DBApp, i.e., the sample application used in Section 4. To this end, we ought to model the interactions among the business components of DBApp as well as the components and processes of the middleware that interact with DBApp. The latter include (and are not limited to) component proxies that marshal and unmarshal parameters of remote method invocations, the transaction manager that coordinates distributed transactions, the a database connectivity driver that facilitates interactions with the database, and the processes for automatic activation and deactivation of objects

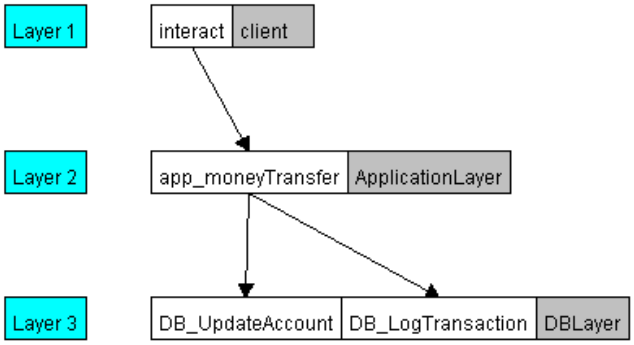

Figure 5: A sample LQN model for DBApp

or components. Thus, although the application has a simple structure, the derivation of the correspondent analysis model becomes very costly.

We believe that this class of issues can be addressed by combining empirical testing and performance modelling according to the following process:

1. The analysis model is built and solved, abstracting from the presence of the middleware. The resulting model will generally have a simple structure.
2. Empirical testing is used to simulate the results of the model (e.g., frequency of operations) on the actual middleware, thus computing how the execution of the middleware and the contention of resources within the middleware affects the performance characteristics of the modelled interactions (e.g., the response time of a given operation may increase because it involves middleware execution).
3. Model parameters are tuned according to the testing results.
4. The process is repeated until the model stabilises.

For instance, Figure 5 shows a Layered Queuing Network (LQN) corresponding to the use-case of Figure 3. A detailed description of LQN models is beyond the scope of this paper, and we refer interested readers to [19]. The layers in Figure 5 represent the main interacting components, i.e., the client, the application and the database. Each component may be present in a number of copies (or threads). White boxes represent the services that each layer provides (limited to services of interest for the considered use-case). Connections between white boxes indicate client-server relationships between services, with arrows pointing to servers. In the specific case represented in the figure, clients interact with the application through the `moneyTransfer` service, which in turn uses services of the database layer to update accounts and log transaction details. Other important parameters of the model that are not indicated in the figure include: the number of calls for each service (for example, both the database services are used twice in the considered case), the CPU and the CPU-time used by each service and the service thinking-times.

Although the middleware is not explicitly represented in the model, it is involved in the execution of each service and affects, for example, the ideal CPU-time and thinking-time. Once empirical measurements are available, the parameters of the LQN model can be tuned accordingly. On the other hand, by solving the model we can compute the frequency of invocations of each service for different numbers of clients. Thus, we can generate the test cases for the middleware accordingly.

The cost of the approach depends on the number of iterations of the process. We expect models to stabilise in a few iterations. However, experimental evidence of this is still missing and further work is required to understand costs and benefits of the integrated approach.

6. CONCLUSIONS AND FUTURE WORK

Distributed component technologies enforce the use of middleware, commercial databases and other off-the-shelf components and services. The software that implements these is available in the initial stages of a software process and moreover it generally embeds the software structures, mechanisms and services that mostly impact the performance in distributed settings. This paper proposed to exploit the early availability of such software to accomplish empirical measurement of performance of distributed applications at architecture-definition-time. To the best of our knowledge, the approach proposed in this paper is novel in software performance engineering.

This paper fulfilled several goals. It discussed the published scientific works related to ours, thus positioning our ideas in the current research landscape. It described a novel approach to performance testing that is based on selecting performance relevant use-cases from the architecture designs, and instantiating and executing them as test cases on the early available software. It indicated important research directions towards engineering such approach, i.e.: The classification of performance-relevant distributed interactions as a base to select architecture use-cases; The investigation of software connectors as a mean to instantiate abstract use-cases on actual deployment technologies and platforms. It reported on experiments that show as the actual performance of a sample distributed application is well approximated by measurements based only on its early available components, thus supporting the main hypothesis of our research. It finally identified the scope of our approach and proposed a possible integration with performance modelling techniques aimed at relaxing its limitations.

Software performance testing of distributed applications has not been thoroughly investigated so far. The reason for this is, we believe, that testing techniques have traditionally been applied at the end of the software process. Conversely, the most critical performance faults are often injected very early, because of wrong architectural choices. Our research tackles this problem suggesting a method and a class of applications such that software performance can be tested in the very early stages of development. In the long term and as far as the early evaluation of middlewares is concerned, we believe that empirical testing may outperform performance estimation models, being the former more precise and easier to use. Moreover, we envision the application of our ideas

to a set of interesting practical cases:

Middleware selection: The possibility of evaluating and selecting the best middleware for the performance of a specific application is reckoned important by many authors, as we already pointed out in Section 2 of this paper. To this end, our approach provides a valuable support. Based on the abstract architecture designs, it allows to measure and compare the performance of a specific application for different middleware and middleware technologies.

COTS selection: A central assumption of traditional testing techniques is that testers have complete knowledge of the software under test as well as of its requirements and execution environment. This is not the case for components off-the-shelf (COTS) that are produced independently and then deployed in environments not known in advance. Producers may fail in identifying all possible usage profiles of a component and therefore testing of the component in isolation (performed by producers) is generally not enough [21]. Limited to the performance concerns, our approach allows to test off-the-shelf components in the context of a specific application that is being developed. Thus, it can be used to complement the testing done by COTS providers and thus assist in selecting among several off-the-shelf components.

Iterative development: Modern software processes prescribe iterative and incremental development in order to control risks linked to architectural choices (see e.g., the Unified Process [6]). Applications are incrementally developed in a number of iterations. During an iteration, a subset of the user requirements is fully implemented. This results in a working slice of the application that can be presently evaluated and, in the next iteration, extended to cover another part of the missing functionality. At the beginning of each iteration, new architectural decisions are generally made whose impact must be evaluated with respect to the current application slice. For performance concerns, our approach can be used when the life cycle architecture is established during the elaboration phase, because it allows to test the expected performance of a new software architecture based on the software that is initially available.

We are now continuing the experiments for augmenting the empirical evidence of the viability of our approach, providing a wider coverage of the possible alternatives of component-based distributed applications. We are also working for engineering the approach, starting from the study of the research problems outlined in this paper.

7. ACKNOWLEDGEMENTS

The authors would like to thank James Skene for his contribution during the preliminary definition of the research ideas presented in this paper.

8. REFERENCES

[1] S. Balsamo, P. Inverardi, and C. Mangano. An approach to performance evaluation of software architectures. In *Proceedings of the First International Workshop on Software and Performance*, pages 178–190, 1998.

[2] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri Nets models. In *Proceedings of the 3rd International Workshop on Software and Performance(WOSP02)*, pages 35–45, 2002.

[3] A. Bertolino, E. Marchetti, and R. Mirandola. Real-time UML-based performance engineering to aid manager's decisions in multi-project planning. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP-02)*, pages 251–261, New York, July 24–26 2002. ACM Press.

[4] S. Bodoff et al. *The J2EE Tutorial*. Addison-Wesley, 2002.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Software Development Process*. Addison-Wesley, 1999.

[7] W. Emmerich. Software Engineering and Middleware: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 117–129. ACM Press, 2000.

[8] W. Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 537–546. ACM Press, 2002.

[9] I. Gorton and A. Liu. Software component quality assessment in practice: successes and practical impediments. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 555–558, New York, 2002. ACM Press.

[10] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman, Reading, MA, USA, 1999.

[11] W. Kao and R. Iyer. A user-oriented synthetic workload generator. In *12th International Conference on Distributed Computing Systems (ICDCS '92)*, pages 270–277. IEEE Computer Society Press, June 1992.

[12] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley Longman, 2000.

[13] C. Lin, A. Avritzer, E. Weyuker, and L. Sai-Lai. Issues in interoperability and performance verification in a multi-orb telecommunications environment. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, pages 567–575, 2000.

[14] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen. Designing a test suite for empirically-based middleware performance prediction. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, 2002. ACS.

[15] N. Medvidovic, E. Dashofy, and R. Taylor. On the role of middleware in architecture-based software

development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4), 2003.

[16] N. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE-00)*, pages 178–187. ACM Press, 2000.

[17] P. Merle. CORBA 3.0 new components chapters. Technical report, TC Document ptc/2001-11-03, Object Management Group, 2001.

[18] H. D. Mills. Top-Down Programming in Large Systems. In R. Ruskin, editor, *Debugging Techniques in Large Systems*. Prentice Hall, 1971.

[19] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[20] R. Pooley. Using UML to derive stocastic process algebra models. In *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW)*, pages 23–34, 1999.

[21] D. Rosenblum. Challenges in exploiting architectural models for software testing. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[22] B. Shannon. Java 2 platform enterprise edition specification, 1.4 - proposed final draft 2. Technical report, Sun Microsystems, Inc., 2002.

[23] J. Skene and W. Emmerich. Model Driven Performance Analysis of Enterprise Information Systems. In *Proc. of the Int. Workshop on Test and Analysis of Component Based Systems, Warsaw, Poland*, volume 82, April 2003.

[24] K. Sreenivasan and A. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, Mar. 1974.

[25] B. Subraya and S. Subrahmanya. Object driven performance testing of Web applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)*, 2000.

[26] E. Weyuker and F. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.