
Performance Testing of Distributed Component Architectures

Giovanni Denaro¹, Andrea Polini², and Wolfgang Emmerich³

¹ Università di Milano-Bicocca, Dipartimento di Informatica Sistemistica e Comunicazione, via Bicocca degli Arcimboldi 8, I-20126 Milano, Italy.
Email: denaro@disco.unimib.it

² Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo," Area di Ricerca del CNR di Pisa, via Moruzzi 1, I-56124 Pisa, Italy.
andrea.polini@isti.cnr.it

³ University College London, Department of Computer Science, Gower Street, WC1E 6BT London, UK.
w.emmerich@cs.ucl.ac.uk

Summary. 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 toward the end of a project. In this chapter, 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 dominates distributed system performance. Drawing on these observations, this chapter presents a novel approach to performance testing of distributed applications. We propose to derive application-specific test cases from architecture designs so that the performance of a distributed application can be tested based on the middleware software at early stages of a development process. We report empirical results that support the viability of the approach.

1 Introduction

Various commercial trends have led 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 is decreasing. Often new services can only be effectively 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 may 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 centralized or client-server architectures but demand the use of distributed software architectures.

In the context of this chapter, we take the perspective of the developer of a component-based system who is interested in devising systematic ways to ascertain that a given distributed software architecture meets the performance requirements of its target 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 report on the weakness of the published scientific literature on *software performance testing* in [27]. 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 [19, 5, 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 majority of research effort has focused on performance analysis models [1, 21, 20, 2, 3, 9] rather than testing techniques. This research shares in general the approach of translating architecture designs, given mostly in the Unified Modeling Language (UML [?]), to models suitable for analyzing performance, such as Layered Queuing Networks (e.g., [20]), Stochastic Petri Nets (e.g., [2]) or stochastic process algebras (e.g., [21]). Estimates of performance are used to reveal flaws in the original

architecture or to compare different architectures and architectural choices. Although models may give useful hints of the performance and help identify bottlenecks, they still 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 parameter is usually guessed through experience, and, as a result, it is not easy to obtain precise models.

With the recent advances in distributed component technologies, such as J2EE [23] and CORBA [18], 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 *middleware*), i.e., middle tier software that provides facilities and services to simplify distributed assembly of components, e.g., communication, synchronization, 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, for example, during the Elaboration Phase of the Unified Process. In this chapter, we argue that this enables performance testing to be successfully applied at an early stage.

The main contribution of this chapter is the description and evaluation of a method for testing performance of distributed software at an early stage of development. 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, middleware 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 chapter 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 the results of an empirical evaluation of the main hypothesis of our research, i.e., that the performance of a distributed application can be successfully measured based on the early available components. Section 5 discusses the limitations of our approach and possible integration with performance modeling techniques. Finally, Section 6 summarizes the contributions of the chapter and outlines 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.

2.1 Performance Testing of Distributed Applications

Some authors have exploited empirical testing for studying the performance of middleware products. Gorton and Liu compare the performance of six different J2EE-based middleware implementations [10]. 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 for increasing numbers 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 et al. investigate the suitability of micro-benchmarks, i.e., lightweight test cases focused on specific facilities of the middleware, 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]), “*incorporating application-specific behavior in to the equation in a simple and practical manner is still an open problem*” and “*it also remains to be seen how far the results from the empirical testing can be generalized across different hardware platforms, databases and operating systems.*” Our research tackles these problems. We study application-specific test cases for early performance evaluation (or also for comparing the performance) of distributed applications in specific deployment environments, which include middleware, 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 [27]. They stress that, given the lack of historical data on the usage of the target system, the architecture is key to identifying 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, which are not covered in the work of Weyuker and Vokolos.

2.2 Software Architecture and Middleware

Medvidovic et al. state the idea of coupling the modeling power of software architectures with the implementation support provided by middleware [15]. They notice that “architectures and middleware address similar problems - large-scale, component-based development - but at different stages of the development lifecycle.” They propose investigating the possibility of defining systematic mappings between architectures and middleware. To this end, they study the suitability of a particular element of software architecture, the *software connector*. Metha et al. propose an interesting classification framework of software connectors [17]. They distinguish among four types of services provided by connectors for enabling and facilitating component interactions: *communication*, i.e., support for transmission of data among components; *coordination*, i.e., support for transfer of control among components; *conversion*, i.e., support for interaction among heterogeneous components; *facilitation*, i.e., support for mediations of the interactions among components (e.g., participation in atomic transactions). A general set of software connector types is identified and classified in the framework, based on the combination of services that they provide. 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 to measure performance attributes of an architecture based on the early available implementation support (of which the middleware is a significant part); Medvidovic et al. aim to build implementation topologies (e.g., bridging of middleware) that preserve the properties of the original architecture. However, the results of previous studies on software connectors and the possibility of mapping architectures on middleware may be important references for engineering our approach, as we further discuss in Sect. 3.2.

3 Approach

In this section, we introduce our approach to early performance testing of distributed component-based software architectures. We also focus on the aspects of the problem that need further investigation. Our long-term goal is to

provide an automated software environment that supports the application of the approach we describe below.

Our performance testing process 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, initialization 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.

3.1 Selecting Performance Use Cases

As it has been noticed by several researchers such as Weyuker [27], the design of test suites for performance testing is radically different from the case of functional testing. In performance testing, the functional details of the test cases, i.e., the actual values of the inputs, are generally of limited 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 durations of tests, the characteristics of the disks, the network bandwidth, and the number and speed(s) of CPU(s). Next, it is important to consider the middleware configuration, for which the table reports parameters in the case of J2EE-based middleware. Here, we do not comment further on workload, physical resource, and middleware parameters, which are extensively discussed in the literature [27, 26, 14].

Other important parameters of performance testing in distributed settings are due to the interactions among distributed components and resources. Different ways of using facilities, services and resources of middleware, and deployment environments are likely to yield different performance results. Performance will differ if the database is accessed many times or rarely. A given middleware may perform adequately for applications that stress persistence and badly for transactions. In some cases, a middleware may perform well or badly for different usage patterns of the same service. The last row of Table 1 classifies some of the relevant interactions in distributed settings according to whether they take place between the middleware and the components, among

Table 1. Performance parameters

Category	Parameter
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	

the components themselves⁴, or to access persistent data in a database. In general, the performance of a particular application will be largely dependent on how the middleware primitives are being used to implement the application's functionality.

We argue that application-specific test cases for performance should be given such that the most relevant interactions triggered specifically by the AUT are covered. According to this principle, the generation of a meaningful test suite for performance testing can be based on either of two possible sources: previously recorded usage profiles or functional cases specified in the early development phases.

The former alternative is viable in cases of system upgrade. In this situation, "histories" of the actual usage profiles of the AUT are likely to be available because of the possibility that they have been recorded in the field. The synthesis of application specific workloads based on recorded usage pro-

⁴Although interactions among distributed components map onto interactions that actually take place at the middleware level, they are elicited at a different abstraction level and thus considered a different category in our classification.

files is a widely studied and fairly well understood research subject in the area of synthetic workload generation (e.g., [11, 25]).

In the case of the development of a completely new application, no recorded usage profile may exist. However, modern software processes tend to define the required functionality of an application under development in a set of scenarios and use cases. To build a meaningful performance test suite, we can associate a weight with each use case and generate a synthetic workload accordingly. The weight should express the importance of each use case in the specific test suite. Obviously, to have a reliable evaluation of the performance characteristics of the application, we need to consider as many use cases as possible. This should be a minor problem because it is often the case that most of the use cases are available in early stages of a software process. For instance, the iterative and incremental development approaches (such as the Unified Software Development Process [5]) demand that the majority of use cases be available at the end of the early process iterations. In such settings, we can therefore assume that the software system developer can use these use cases to derive test cases to evaluate the performance of the final application before starting with the implementation phase. On the basis of the results obtained, the developer can eventually revise the decisions taken in order to obtain better “expected” performance. To this end, several possibilities are available at this stage (less expensive with respect to late system refactoring, which may be required due to poor performance), such as a revision of the architecture or a recalibration of some choices concerning the middleware configuration.

3.2 Mapping Use Cases to Middleware

At the initial stages of the software process, software architectures are generally defined at a very abstract level. The early use-cases focus on describing the business logic while they abstract the details of the deployment platform and technology. One of the strengths of our approach is indeed the possibility of guiding software engineers through the intricate web of architectural choices, off-the-shelf components, distributed component technologies, middleware, and deployment options, while keeping the focus on the performance of the final product. The empirical measurements of performance may provide the base for comparing possible alternatives. Consequently, to define a performance test case, the abstract use-cases must be augmented with the following information:

- The mapping between the early available components (if any) and the components represented in the abstract use-cases;
- The distributed component technology and the actual middleware with respect to which the performance test is to be performed;
- The characteristics of the deployment of the abstract use-cases on the actual middleware platform, i.e., the specification of how the described com-

ponent interactions take place through the selected component technology and middleware.

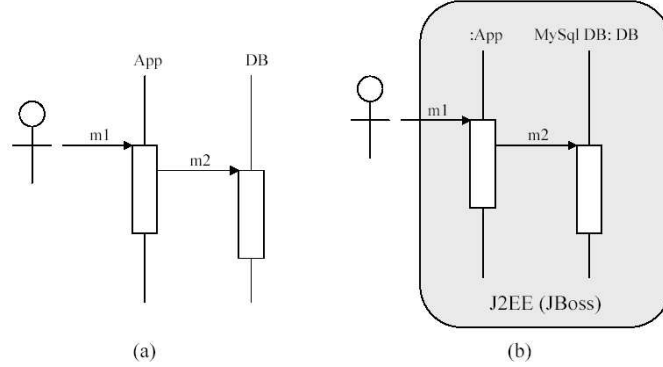


Fig. 1. A sample use-case (a) and part of a corresponding performance test case (b)

The two former requirements can be trivially addressed. For example, Fig. 1 (a) illustrates a sample abstract use-case in which an actor accesses the service `m1` provided by the component `App`, which in turn uses the service `m2` provided by the component `DB`. Correspondingly, Fig. 1 (b) illustrates a performance test case in which the component `DB` is referenced as the available MySQL database engine, while the component `App` is not early available; the whole application is deployed using the J2EE component technology and the JBoss application server as middleware. The rest of this section discusses the problem of specifying the deployment characteristics.

At the architectural level, the properties of the component interactions can be described in terms of *software connectors*⁵. Recent studies (e.g., [15]) have investigated the role that software connectors may play in software design, showing that they may relevantly contribute to bridging the gap between the high-level application view of a software architecture and the implementation support provided by distributed component technologies and middleware. [17] attempts to classify software connectors and identifies a general set of *connector types*, their characteristics (*dimensions*), and the possible practical alternatives for each characteristic (*values*). For instance, the `procedure call` is identified as a connector type that enables communication and coordination among components; `synchronicity` is one of the dimensions of a

⁵This is the spirit of the definition of software connectors given by Shaw and Garlan [?]: *connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.*

procedure call connector; and `synchronous` and `asynchronous` are the possible values of such a dimension. When all dimensions of a connector type are assigned to specific values, the resulting instance of the connector type identifies a connector *species*, e.g., the `remote method invocation` can be considered as a species of the procedure call connector type. Our approach to the specification of the deployment characteristics leverages and extends the connector taxonomy of [17].

Up to now we have identified an initial set of connector types that specifically apply to the case of component interactions that take place through J2EE-compliant middleware. Giving values to the dimensions of these connectors allows for specifying the characteristics of the deployment of an abstract use-case on an actual middleware platform based on the J2EE specification. Specifically, we identified the following connector types: J2EE remote service, J2EE distributor, J2EE arbitrator, and J2EE data access.

The *J2EE remote service* connector extends and specializes the procedure call connector type of [17]. This connector specifies the properties of the messages that flow among interacting components. We identified the following relevant dimensions for this connector:

- **Synchronicity:** A remote service can be either synchronous or asynchronous. Specifying a value for the synchronicity dimension allows to decide if the service is meant to be implemented as a synchronous method invocation or as an asynchronous event propagation.
- **Parameters:** This dimension specifies the number of parameters and their expected sizes in bytes. This allows for simulating the dependencies between performance and the transfer of given amounts of data between components. Moreover, if the component that provides the service is one of the early available components, types and values of the parameters must also be provided to perform the actual invocation during the test. In this case, if the service is expected to be invoked a number of times during the test, we can embed in the connector a strategy for choosing the values of the parameters:
 1. a single value may be given. This value will be used every time the service is invoked during the test;
 2. a list of values may be given. Each time the service is invoked, a value of the list is sequentially selected;
 3. a list of values and an associated probability distribution may be given. Each time the service is invoked, a value of the list is selected by sampling the distribution.

The *J2EE distributor* connector extends and specializes the distributor connector type of [17]. This connector allows us to specify the deployment topology. We identified the following relevant dimensions for this connector:

- **Connections:** This dimension specifies the properties of the connections among the interacting components, i.e., the physical hosts on which they

are to be deployed in the testing environment and the symbolic names used to retrieve the component factories through the naming service.

- **Types.** This dimension specifies the (expected) implementation type of the interacting components. Possible values are client application, session bean, entity bean⁶, and database table.
- **Retrieving.** This dimension specifies how to use the component factories (for components and interactions to which this is applicable) for retrieving references to components. In particular, either the default or finder method can be specified (non-standard retrieving methods of component factories are called *finders* in the J2EE terminology).

The *J2EE arbitrator* connector extends and specializes the arbitrator connector type of [17]. This connector specifies the participation in transactions and the security attributes of the component interactions. We identified the following relevant dimensions for this connector:

- **Transactions:** This dimension specifies the participation of a component interaction in transactions. Possible values are none, starts, and participates; *none* if the interaction does not participate in any transaction; *starts* if the interaction starts a new, possibly nested, transaction; *participates* if the interaction participates in the transaction of the caller.
- **Security:** This dimension specifies the security attributes of a component interaction. In particular, it specifies if services can be accessed by all users, specific users, or specific user groups, and which component is responsible for authentication in the two last cases.

The *J2EE data access* connector extends and specializes the data access connector type of [17]. This connector mediates the communication between J2EE components and a database, specifying the structure of the database and how the interactions are handled. In particular, we identified the following relevant dimensions for this connector:

- **Tables:** This dimension specifies characteristics of the tables and their respective fields in the database.
- **Relationships:** This dimension specifies the presence of relationships among the tables in the database.
- **Management:** In J2EE components, persistence can be handled either by implementing the access functions (e.g., queries) in the component code (this is called bean managed persistence, or BMP) or by using a standard mechanism embedded in the middleware (this is called container managed persistence, or CMP).

⁶Session beans are J2EE components that provide business services. Thus, session beans are often used as the interface between J2EE applications and client applications. Entity beans are J2EE components that represent persistent data within an application. Each database table is generally associated with an entity bean. The data in the entity bean are synchronized with the database. Thus, entity beans are often used as the interface between J2EE applications and databases.

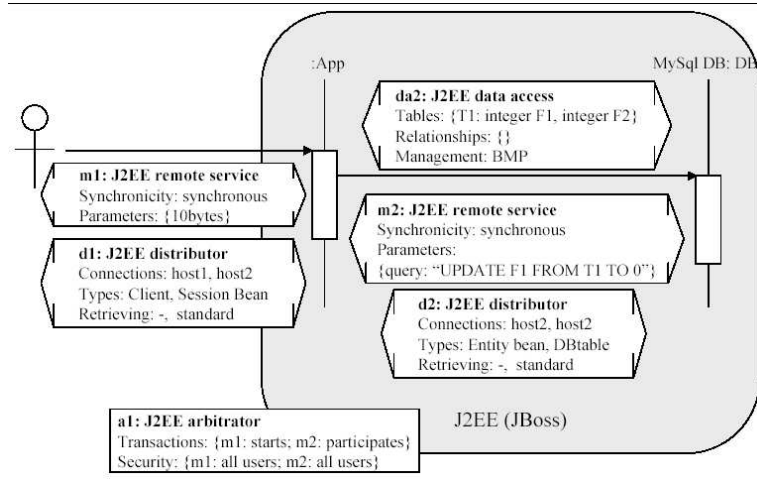


Fig. 2. A performance test case associated with the use-case in Fig. 1

Figure 2 illustrates the application of connectors to the sample use-case of Fig. 1. As specified by the J2EE remote service connectors, the interactions `m1` and `m2` are both synchronous (i.e., they are assumed to be remote method invocations) and have just one input parameter. In the case of `m1`, only the parameter size is relevant since the server component `App` is not available early. Conversely, in the case of `m2`, the actual value of the parameter is needed, since the database is available. The parameter specified is the actual SQL code to be executed on the database, and the “single value” strategy is used. The database structure assumed is specified in the J2EE data access connector `da2`, and consists of a table (`T1`) with two integer fields (`F1` and `F2`) and no relationship, while the interactions between the component `App` and the MySQL database are supposed to follow the bean managed persistence paradigm. The two J2EE distributor connectors, `d1` and `d2`, specify that the component `App` and the database are deployed on the same host (`host2`), while the client is deployed on a different host (`host1`). The interface between the client and the component `App` is provided by a session bean EJB component, and the interface between `App` and the database is handled by an entity bean EJB component. The retrieving strategy, when applicable, uses the standard methods provided by the platform. Finally, the J2EE arbitrator connector specifies that `m1` starts a transaction in which `m2` participates, and no special security policy is considered. The information given in Fig. 2 identifies a specific performance test case associated with the use-case in Fig. 1.

Notice that Fig. 2 is meant just for exemplification purposes, and not to suggest an approach in which use-case diagrams must be annotated with connector information before testing. In a mature and engineered version of our approach, we envision the possibility that a tool analyzes the abstract use-cases and extracts a simple list of alternatives for each interaction dimen-

sion. The performance engineer would then have the choice of selecting the best suited alternatives according to the performance requirements, or testing different alternatives to find out the one that works best (in a sort of what-if-analysis fashion). Software connectors provide the reasoning framework towards this goal. Furthermore, our current knowledge about all connector types needed and their dimensions is limited because it is based on a simple case in which we have experimented with the application of the approach (Sect. 4 gives the details of this). We believe that we are on the right path, even though we are aware that further work is still needed to understand the many dimensions and species of software connectors and their relationships with the deployment technologies and platforms possible.

3.3 Generating Stubs

So far, we have suggested that early test cases of performance can be derived from 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 their 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 adjusted specifically to use-cases, i.e., different use-cases will require different stubs of the same component. Stubs will ensure only that the distributed interactions happen as specified and that the other components are coherently exercised. Our idea of the engineered approach is that the required stubs are automatically generated based on the information contained in use-case elaborations and software connectors. For example, referring once again to Fig. 2, if the component *App* is not available, its stub would be implemented such that it is able to receive the invocations of the service *m1* and, consequently, invoke the service *m2* through the actual middleware. The actual SQL code embedded in the remote service connector of *m2* would be hardcoded in the stub. As for *m1*, it would contain empty code for the methods, but set the corresponding transaction behavior as specified. Of course, many functional details of *App* are generally not known and cannot be implemented in the stub. Normally, this will result in discrepancies between execution times of 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 derives from the observation that the available components, e.g., middleware and databases, embed software that mainly impacts performance. The coupling between such implementation support and

the application-specific behavior 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 of orders of magnitude less than the impact of the interactions facilitated by middleware and persistence technologies, such as databases. We report an initial empirical assessment of this hypothesis in Sect. 4 of this chapter, but are aware that further empirical studies are needed.

The generation of the fake version can be made easier if we can use UML to describe the software architecture. The use of UML in fact enables the use of all UML-based tools. An interesting investigation in this direction can be found in [16]. In this work the authors propose different techniques to introduce concepts as connectors and architectural styles as first order concepts inside an “extended” fully compliant UML.

3.4 Executing the Test

Building 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 mapping the use cases to deployment technologies and platforms, and generating stubs to replace missing components. Also, we must automate deployment and implementation of workload generators, initialization of persistent data, execution of measurements and reporting of results.

In particular, workload generators can be characterised in several different ways, and many different workloads can be found in the literature (e.g. [6, 24]). It is a developer’s duty to choose the one that best represents the load that it expects for the application during normal use. Then, after the workload has been chosen, for instance, from a list of different possible choices, and the probability distributions have been associated with the relevant elements in that workload, it is possible to automatically generate the corresponding “application client” that generates invocations according to the chosen workload type and distributions.

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 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 Sect. 3. Finally, we executed the performance test

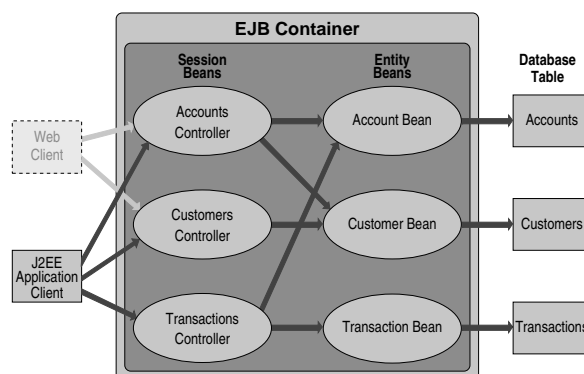


Fig. 3. The Duke's Bank application

(with different numbers of application clients) on the early available components and compared the results with the performance measured on the actual application.

4.1 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 Microsystems 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 are 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 sized component-based distributed applications. The Duke's bank application is referred to as DBApp in the rest of this chapter.

The organization of the DBApp is given in Fig. 3 (borrowed from [4]). The application can be accessed by both Web and application clients. It consists of six EJB (Enterprise Java Bean [23]) components that handle operations issued by the users of a hypothetical bank. The six components can be associated with classes of operations that are related to bank accounts, customers, and transactions. For each of these classes of operations, a pair of session bean and entity bean is provided. Session beans are responsible for the interface for users and entity beans handle the mapping of components to the underlying database table. The arrows represent possible 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 4 illustrates the selected use-case in UML.

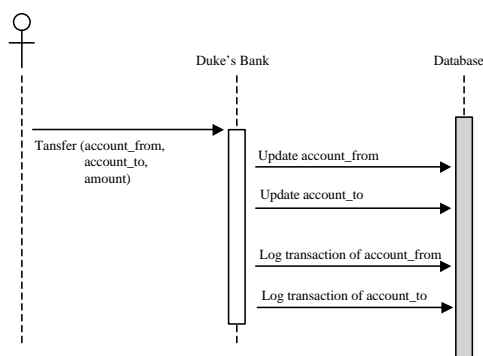


Fig. 4. A sample use-case for the Duke's Bank

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 to store 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 the database engine available early in the software process.

Thus, for the test, we used the same database engine, table structure, and SQL code as in the original application. This is why we represent the database as a shadowed box in the figure. Unlike the database, the business components of the DBApp are assumed not to be available; thus we had to generate corresponding stubs.

For implementing the stubs, we had to map the abstract use-case onto the selected deployment technology, i.e., J2EE. We already commented on the role that software connectors may play in the mapping. 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 participate in the calling transaction and embed the actual SQL code. We set up the database factory such that the database connection is initialized for each call⁷. The DBApp uses entity beans and bean managed persistence to handle the interactions with the database tables. Based on this information, we implemented the stubs as needed to realize the interactions in the use-case considered and we deployed the test version of the DBApp (referred to as DBTest) on the JBoss application server.

⁷Although this may sound like a bad implementation choice, we preferred to maintain the policy of the original application to avoid biases in the comparison.

Finally, we implemented a workload generator and initialized 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 case in which a group of people is paying the same authority over the Internet. Incidentally, we notice that, in an automated test environment, initialization of persistent data would require to specify only the performance sensitive part of the information, with the actual values in the database tables being of little importance. For example, in our case, only the number of elements in each table and their relationships with the considered use-case, i.e., whether each clients accesses 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, from one to one hundred. The two applications were deployed on a JBoss 3.0 application server running on a PC equipped with a 1 GHz Pentium III CPU, 512 MB of RAM, and the Linux operating system. To generate the workload, we run the clients on a Sun Fire 880 equipped with four 850 MHz Sparc CPUs 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 concurrent existence of a lot of active session beans, we restarted the application server between two successive experiments. JBoss was run using the default configuration. Finally, the specific setting concerning the particular use case, as already discussed in the previous paragraphs, saw the use of remote method calls between the components and the use of the transaction management service to handle the data shared by the various beans consistently.

4.2 Empirical Results

We executed both DBApp and DBTest for increasing numbers of clients, and measured the latency time for the test case. We repeated each experiment 15 times and measured the average latency time. Figure 5 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 σ . The ratio between σ and the expectations are, in fact, definitively lower than 0.15, for both the DBApp and the DBTest.

The results of this experiment suggest the viability of our research because they prove that the performance of the DBApp in a specific use-case is well

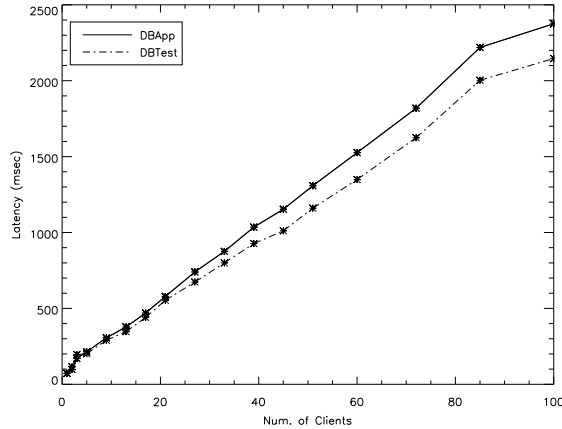


Fig. 5. Latency of DBApp and DBTest for increasing numbers of clients

approximated by the DBTest, which is made of early available components. However, although the first results are encouraging, we are aware that a single experiment cannot be generalized. 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 calls, J2EE-based application servers other than JBoss, CORBA-based middleware, and other commercial databases and in the presence of other early available components.

5 Scope and Extensions

Our results support 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, synchronization, persistence of data, transactions, load balancing, and threading policies. Thus, in most cases critical contentions 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 what 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 middleware (e.g., [10, 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 generalized to test all components that are available at the beginning of the development process, for example, components acquired off-the-shelf from 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 of discovering the bottleneck, since it would not be exhibited by a stub of the component. Performance analysis models remain the primary reference for evaluation of performance in such cases.

Currently, we are studying the possibility of combining empirical testing and performance modeling, aimed 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 problems of applying performance analysis to middleware-based distributed systems is that 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 Sect. 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 or components. Thus, although the application has a simple structure, the derivation of the corresponding analysis model becomes very costly.

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

1. The analysis model is built and solved, abstracting from 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 affect the performance characteristics of the modeled interac-

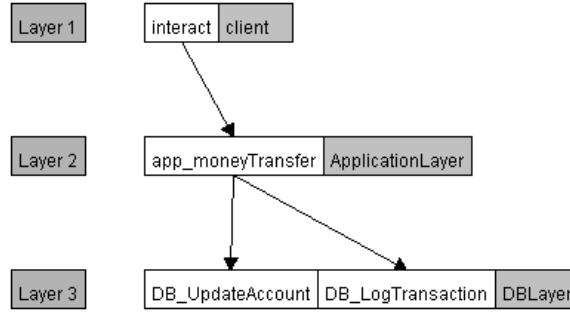


Fig. 6. A sample LQN model for DBApp

tions (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 stabilizes.

For instance, Fig. 6 shows a Layered Queuing Network (LQN) corresponding to the use-case of Fig. 4. A detailed description of LQN models is beyond the scope of this chapter, and we refer interested readers to [20]. The layers in Fig. 6 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 to the use-cases considered). 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 stabilize 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 chapter set out 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 chapter is novel in software performance engineering.

This chapter 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 early available software. It indicated important research directions toward engineering such an approach, i.e., the classification of performance relevant distributed interactions as a base to select architecture use-cases and the investigation of software connectors as a means to instantiate abstract use-cases on actual deployment technologies and platforms. It reported experiments that showed that 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 modeling techniques aimed at relaxing its limitations.

Software performance testing of distributed applications has not been thoroughly investigated so far. The reason for this, we believe, is that testing techniques have traditionally been applied at the end of the software process. Conversely, the most critical performance faults are often present 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 middleware is concerned, we believe that empirical testing may outperform performance estimation models, since the former is 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 Sect. 2 of this chapter. To this end, our approach provides a valuable support. Based on abstract architecture designs, it allows us to measure and compare the performance of a specific application with different 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 off-the-shelf components (COTS) that are produced independently and then deployed in environments not known in advance. Vendors may fail in identifying all possible usage profiles of a component, and therefore the testing of the component in isolation (performed by vendors) is not generally enough [22]. Limited to performance concerns, our approach allows us 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 from 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 [5]). Applications are developed incrementally, through a number of iterations. During each iteration, a subset of the user requirements is fully implemented. This results in a working slice of the application that can be 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 made whose impact must generally 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 us to test the expected performance of a new software architecture based on the software that is initially available.

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

Acknowledgments

This work has been partially funded through the EU IST project SEGRAVIS.

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 Software Development Process*. Addison-Wesley, 1999.
6. Vittorio Cortellessa and Raffaella Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*, 44:101–129, 2002.
7. W. Emmerich. Software engineering and middleware. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 117–132. 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. W. Emmerich and J. Skene. Model driven performance analysis of enterprise information systems. In *Proceedings of the International Workshop on Testing and Analysis of Component-Based Systems (TACOS'03)*, 2003.
10. 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.
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. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.
17. 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.
18. P. Merle. CORBA 3.0 new components chapters. Technical report, TC Document ptc/2001-11-03, Object Management Group, 2001.
19. H. D. Mills. Top-Down Programming in Large Systems. In R. Ruskin, editor, *Debugging Techniques in Large Systems*. Prentice Hall, 1971.

20. 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.
21. R. Pooley. Using UML to derive stochastic process algebra models. In *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW)*, pages 23–34, 1999.
22. 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.
23. B. Shannon. Java 2 platform enterprise edition specification, 1.4 - proposed final draft 2. Technical report, Sun Microsystems, Inc., 2002.
24. J. Skene and W. Emmerich. A model-driven approach to non-functional analysis of software architectures. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE2003)*, pages 236–239, 2003.
25. K. Sreenivasan and A. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, March 1974.
26. 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.
27. 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.