Performance Testing of Distributed Component Architectures

Giovanni Denaro[1], Andrea Polini[2], and Wolfgang Emmerich[3]

[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. Email: `andrea.polini@isti.cnr.it`

[3] University College London, Department of Computer Science, Gower street, WC1E 6BT London, UK. Email: `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 towards 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 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.

In the context of this chapter, we take the perspective of the producer 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 their 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 reported on the weakness of the published scientific literature on *software performance testing* in [29]. 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 [20, 6, 13], 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 efforts has focused on performance analysis models [1, 22, 21, 2, 3, 10] 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. [21]), Stochastic Petri Nets (e.g. [2]) or stochastic process algebras (e.g. [22]). 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 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 [24] and CORBA [19], distributed systems are no longer built from scratch [8]. 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, synchronisation, threading and load balancing facilities and transaction and security management services [9]. 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 in 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 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 chapter 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.

2.1 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 middleware implementations [11]. 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 [14]. Liu et al. investigate the suitability of micro-benchmarks, i.e., light-weight test cases focused on specific facilities of the middleware, such as, directory service, transaction management and persistence and security support [15]. 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 ([15]), “how to incorporate 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 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 [29].

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.

2.2 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 [16]. 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 middleware. To this end, they study the suitability of a particular element of software architecture, the *software connector*. Metha, Phadke and Medvidovic propose an interesting classification framework of software connectors [18]. They distinguish among four types of services provided by connectors for enabling and facilitating component interactions: *Communication*, i.e., support to the transmission of data among components; *Coordination*, i.e., support to transfer of control among components; *Conversion*, i.e., support to the interaction among heterogeneous components; *Facilitation*, i.e., support to 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 at measuring performance attributes of an architecture based on the early available implementation support (which the middleware is a significant part of); Medvidovic et al aim at building 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 Section 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, 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.

3.1 Selecting Performance Use Cases

As it has been noticed by several researchers, such as Weyuker [29], 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 duration of tests, 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 middleware. Here, we do not comment further on workload, physical resource and middleware parameters, which are extensively discussed in the literature [29, 28, 15].

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 quite 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 com-

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

ponents, among the components themselves[4] 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 specifically triggered 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-

[4]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.

files is a widely studied and fairly well understood research subject in the area of synthetic workload generation (e.g.[12, 27]).

When the development of a completely new application is the case, 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 to 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 [6]) 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 base of the obtained results the developer can eventually revise the taken decisions in order to obtain better "expected" performance. To this end, several possibilities are available at this stage, (at a less expensive costs with respect to a late system refactoring, which might be required due to poor performance), such as, a revision of the architecture or a "re-"calibration of some choices concerning the middleware configuration.

3.2 Mapping Use Cases to Middleware

In 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 driving software engineers through the intricate web of architectural choices, off-the-shelf components, distributed component technologies, middleware and deployment options, keeping the focus on the performance of the final product. The empirical measurements of performance may provide the base for comparing the 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-

mponent technology

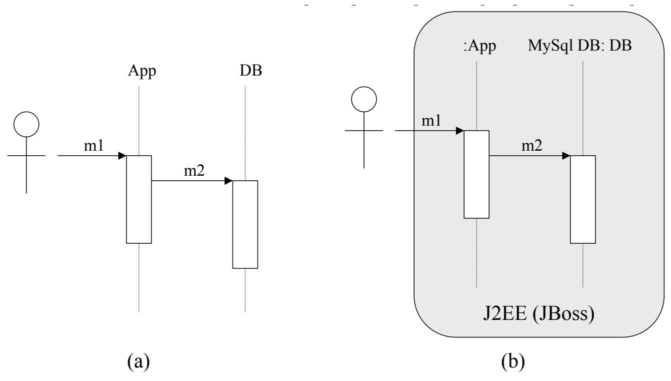

Fig. 1. An sample use-case (a) and part of a corresponding performance test case (b)

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 instanced 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., [16]) have investigated the role that software connectors may play in software design, showing that they may relevantly contribute to bridge the gap between the high-level application view of a software architecture and the implementation support provided by distributed component technologies and middleware. [18] 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 procedure call connectors; and `synchronous` and `asynchronous` are the possible values

[5]This is, for example, the spirit of the definition of software connectors given by Shaw and Garlan [25]: *connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.*

of such dimension. When all dimensions of a connector type are assigned to specific values, the resulting instance of the connector type identifies a connector *specie*, e.g., the `remote method invocation` can be considered as a specie of the procedure call connector type. Our approach to the specification of the deployment characteristics leverages and extends the connector taxonomy of [18].

Up to now, we identified an initial set of connector types that specifically apply to the case of component interactions that take place through a 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 specialises the procedure call connector type of [18]. 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 select if the service must be instanced as a synchronous method invocation or as an asynchronous event propagation, respectively.
- Parameters: This dimension specifies the number of parameters and their expected size in bytes. This allows for simulating the dependences between performance and the transfer of given amounts of data among components. Moreover, if the component that provides the service is one of the early available components, also types and values of the parameters must be provided to perform the actual invocation during the test. In this latter 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 sampling the distribution.

The *J2EE distributor* connector extends and specialises the distributor connector type of [18]. This connector allows 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 bean[6] and database table.
- Retrieving. This dimension specifies how to use the component factories (for components and interactions this is applicable to) 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 specialises the arbitrator connector type of [18]. 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 in transactions of a component interaction. Possible values are: none, starts and participates: none, if the interaction does not participate in any transaction; starts, if the interaction starts a new, possible 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 such two latter cases.

The *J2EE data access* connector extends and specialises the data access connector type of [18]. 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 implementing the access functions (e.g., queries) in the component code (this is called bean managed persistence, BMP) or using standard mechanism embedded in the middleware (this is called container managed persistence, CMP).

[6]Session 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 to an entity bean. The data in the entity bean are taken synchronised with the database. Thus, entity bean are often used as the interface between J2EE applications and databases.

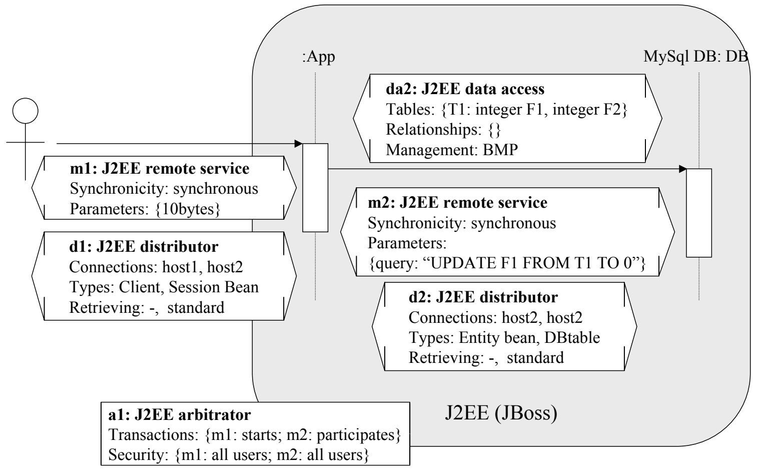
:App
MySql DB: DB
da2: J2EE data access
Tables: {T1: integer F1, integer F2}
Relationships: {}
Management: BMP
m1: J2EE remote service
Synchronicity: synchronous
Parameters: {10bytes}
m2: J2EE remote service
Synchronicity: synchronous
Parameters:
{query: "UPDATE F1 FROM T1 TO 0"}
d1: J2EE distributor
Connections: host1, host2
Types: Client, Session Bean
Retrieving: -, standard
d2: J2EE distributor
Connections: host2, host2
Types: Entity bean, DBtable
Retrieving: -, standard
a1: J2EE arbitrator
Transactions: {m1: starts; m2: participates}
Security: {m1: all users; m2: all users}
J2EE (JBoss)

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

Fig. 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 worth it, being the server component App not early available. Conversely, in the case of m2, also the actual value of the parameter is needed, being the database available. The specified parameter is the actual SQL code to be executed on the database and the "single value" strategy is used. The assumed database structure 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 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 purpose 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 analyses the abstract use-cases and extracts the simple list of alternatives for each interaction dimension. The performance engineer would then have the choice of selecting the best suited alternatives according to the performance requirements or test different alternatives to find out the one that works best (in a sort of what-if-analysis fashion). Software connectors provides the reasoning framework towards this goal. Furthermore, our current knowledge about all needed connector types and their dimensions is limited because it is based on a simple case in which we have experimented the application of the approach (Section 4 gives the details of this initial experience). 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 possible deployment technologies and platforms.

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 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 Fig. 2, if the component *App* is not available, its stub would be implemented such that it is just able to receive the invocations of the service `m1` and consequently invokes the service `m2`, through the actual middleware. The actual SQL code embedded in the remote service connector of `m2` would be hard-coded in the stub. As for `m1`, it would contain empty code for the methods, but set the corresponding transaction behaviour 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 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., middleware 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 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 enables, in fact, the use of all the UML-based tools. A first interesting investigation in this direction can be found in [17]. In this work the authors propose different techniques to introduce concepts as connectors and architectural styles as a first order concepts inside an "extended" fully conform UML.

3.4 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 mapping the use cases 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.

In particular workload generator can be characterised in several different way, and many different workload can be found in literature (e.g.[7, 26]). It is a developer duty to choose the one that better represent the load that it expects for the application during the normal usage. Then after that the type of workload have been chosen, for instance from a list of possible different choice, and that the probability distributions have been associated to the relevant elements in the particular workload, it is possible to automatically generate the corresponding "application client" that generate 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 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

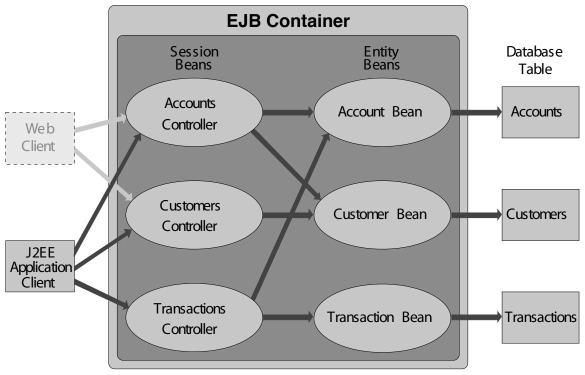

Fig. 3. The Duke's Bank application

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 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 chapter.

The organisation 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 Beans [24]) 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 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. Fig. 4 illustrates the selected use-case in UML.

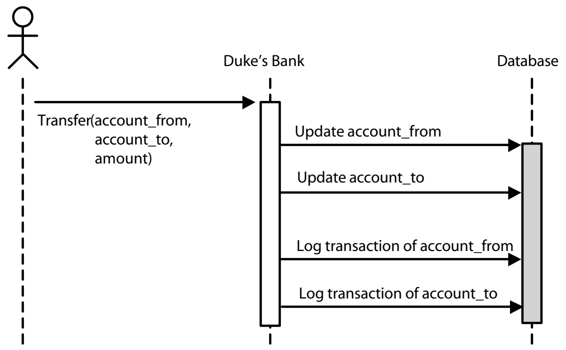
Duke's Bank
Database
Transfer(account_from, account_to, amount)
Update account_from
Update account_to
Log transaction of account_from
Log transaction of account_to

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 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-case on 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 to the calling transaction and embed the actual SQL code; we set up the database factory such that the database connection is initialised for each call[7]; 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 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

[7]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.

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 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.

4.2 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. Fig. 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. 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

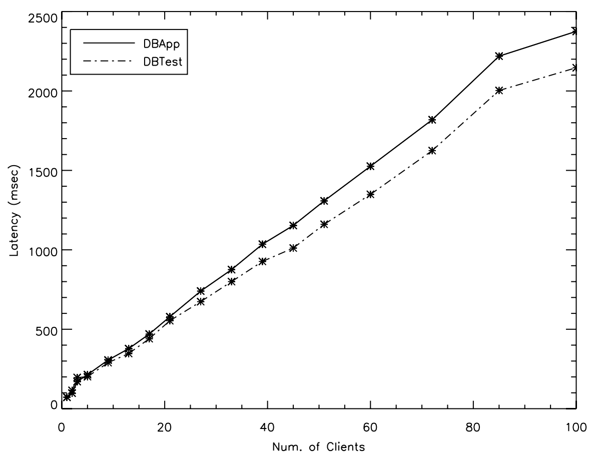

Fig. 5. Latency of DBApp and DBTest for increasing numbers of clients

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 middleware, 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 middleware (e.g., [11, 14, 15]), 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 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.

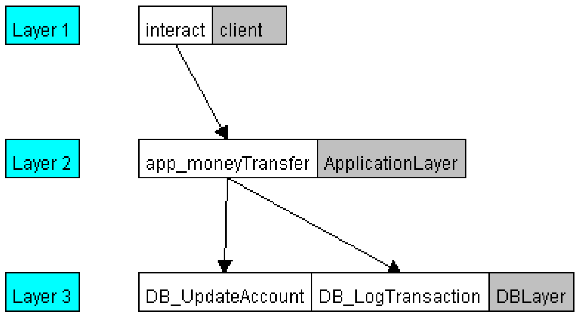

Fig. 6. A sample LQN model for DBApp

4. The process is repeated until the model stabilises.

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 [21]. 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 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 chapter 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 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 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 middleware 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 chapter. 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 [23]. 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 and 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 chapter.

Acknowledgments

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

References

1. Balsamo S, Inverardi P, Mangano C (1998) An approach to performance evaluation of software architectures. In: Proceedings of the First International Workshop on Software and Performance
2. Bernardi S, Donatelli S, Merseguer J (2002) From UML sequence diagrams and statecharts to analysable Petri Nets models. In: Proceedings of the 3rd International Workshop on Software and Performance(WOSP02)

3. Bertolino A, Marchetti E, Mirandola R. (2002) 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). ACM Press
4. Bodoff S et al. (2002) The J2EE Tutorial. Addison-Wesley
5. Booch G, Rumbaugh J, Jacobson I (1999) The Unified Modeling Language User Guide. Addison-Wesley
6. Booch G, Rumbaugh J, Jacobson I (1999) The Unified Software Development Process. Addison-Wesley
7. Cortellessa V, Mirandola R (2002) PRIMA-UML: a performance validation incremental methodology on early UML diagrams. Science of Computer Programming 44:101–129
8. Emmerich W (2000) Software engineering and middleware. In: Proceedings of the 22th International Conference on Software Engineering (ICSE-00). ACM Press
9. Emmerich W (2002) Distributed component technologies and their software engineering implications. In: Proceedings of the 24th International Conference on Software Engineering (ICSE-02). ACM Press
10. Skene J, Emmerich W (2003) Model driven performance analysis of enterprise information systems. In: Proceedings of the International Workshop on Testing and Analysis of Component-Based Systems(TACOS'03)
11. Gorton I, Liu A (2002) Software component quality assessment in practice: successes and practical impediments. In: Proceedings of the 24th International Conference on Software Engineering (ICSE-02). ACM Press
12. Kao W, Iyer R (1992) A user-oriented synthetic workload generator. In: Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS'92). IEEE Computer Society Press
13. Kruchten P (2000) The Rational Unified Process: An Introduction. Addison-Wesley
14. Lin C, Avritzer A, Weyuker E, Sai-Lai L (2000) 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)
15. Liu Y, Gorton I, Liu A, Jiang N, Chen S (2002) Designing a test suite for empirically-based middleware performance prediction. In: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'02)
16. Medvidovic N, Dashofy E, Taylor R (2003) On the role of middleware in architecture-based software development. International Journal of Software Engineering and Knowledge Engineering 13(4)
17. Medvidovic N, Rosenblum D, Redmiles D, Robbins J (2002) Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology 11(1):2–57
18. Mehta N, Medvidovic N, Phadke S (2000) Towards a taxonomy of software connectors. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE-00). ACM Press
19. Merle P (2001) CORBA 3.0 new components chapters. Technical report, TC Document ptc/2001-11-03, Object Management Group
20. Mills H D (1971) Top-Down Programming in Large Systems. In: Ruskin R ed, Debugging Techniques in Large Systems. Prentice Hall

21. Petriu D, Shousha C, Jalnapurkar A (2000) Architecture-based performance analysis applied to a telecommunication system. IEEE Transactions on Software Engineering 26(11):1049–1065
22. Pooley R (1999) Using UML to derive stochastic process algebra models. In: Proceedings of the 15th UK Performance Engineering Workshop (UKPEW)
23. Rosenblum D (1998) 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). ACM Press
24. Shannon B (2002) Java 2 platform enterprise edition specification, 1.4 - proposed final draft 2. Technical report, Sun Microsystems Inc.
25. Shaw M, Garlan D (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall
26. Skene J, Emmerich W (2003) A model-driven approach to non-functional analysis of software architectures. In: Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE2003)
27. Sreenivasan K, Kleinman A (1974) On the construction of a representative synthetic workload. Communications of the ACM 17(3):127–133
28. Subraya B, Subrahmanya S (2000) Object driven performance testing of Web applications. In: Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)
29. Weyuker E, Vokolos F (2000) Experience with performance testing of software systems: issues, an approach, and case study. IEEE Transactions on Software Engineering 26(12):1147–1156