Software Performance Testing

Xiang Gan

Helsinki 26.09.2006

Seminar paper

University of Helsinki

Department of Computer Science

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section	Laitos – Institution – Department
Faculty of Science	Department of Computer Science

Tekijä – Författare – Author

Xiang Gan

Työn nimi – Arbetets titel – Title

Software performance testing

Oppiaine – Läroämne – Subject

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
	26.9.2006	9

Tiivistelmä – Referat – Abstract

Performance is one of the most important aspects concerned with the quality of software. It indicates how well a software system or component meets its requirements for timeliness. Till now, however, no significant progress has been made on software performance testing. This paper introduces two software performance testing approaches which are named workload characterization and early performance testing with distributed application, respectively.

ACM Computing Classification System (CCS):
A.1 [Introductory and Survey],
D.2.5 [Testing and Debugging]

Avainsanat – Nyckelord – Keywords

software performance testing, performance, workload, distributed application

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Contents

1 Introduction 1

2 Workload characterization approach 2

2.1 Requirements and specifications in performance testing 2
2.2 Characterizing the workload 2
2.3 Developing performance test cases 3

3 Early performance testing with distributed application 4

3.1 Early testing of performance 5
3.1.1 Selecting performance use-cases 5
3.1.2 Mapping use-cases to middleware 6
3.1.3 Generating stubs 7
3.1.4 Executing the test 7

4 Conclusion 8

References 9

1 Introduction

Although the functionality supported by a software system is apparently important, it is usually not the only concern. The various concerns of individuals and of the society as a whole may face significant breakdowns and incur high costs if the system cannot meet the quality of service requirements of those non-functional aspects, for instance, performance, availability, security and maintainability that are expected from it.

Performance is an indicator of how well a software system or component meets its requirements for timeliness. There are two important dimensions to software performance timeliness, responsiveness and scalability [SmW02]. Responsiveness is the ability of a system to meet its objectives for response time or throughput. The response time is the time required to respond to stimuli (events). The throughput of a system is the number of events processed in some interval of time [BCK03]. Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demand for the software function increases [SmW02].

As Weyuker and Vokolos argued [WeV00], usually, the primary problems that projects report after field release are not system crashes or incorrect systems responses, but rather system performance degradation or problems handling required system throughput. If queried, the fact is often that although the software system has gone through extensive functionality testing, it was never really tested to assess its expected performance. They also found that performance failures can be roughly classified as the following three categories:

- the lack of performance estimates,
- the failure to have proposed plans for data collection,
- the lack of a performance budget.

This seminar paper concentrates upon the introduction of two software performance testing approaches. Section 2 introduces a workload characterization approach which requires a careful collection of data for significant periods of time in the production environment. In addition, the importance of clear performance requirements written in requirement and specification documents is emphasized, since it is the fundamental basis to carry out performance testing. Section 3 focuses on an approach to test the performance of distributed software application as early as possible during the entire software engineering process since it is obviously a large overhead for the development team to fix the performance problems at the end of the whole process. Even worse, it may be impossible to fix some performance problems without sweeping redesign and re-implementation which can eat up lots of time and money. A conclusion is made at last in section 4.

2 Workload characterization approach

As indicated [AvW04], one of the key objectives of performance testing is to uncover problems that are revealed when the system is run under specific workloads. This is sometimes referred to in the software engineering literature as an operational profile [Mus93]. An operational profile is a probability distribution describing the frequency with which selected important operations are exercised. It describes how the system has historically been used in the field and thus is likely to be used in the future. To this end, performance requirement is one of the necessary prerequisites which will be used to determine whether software performance testing has been conducted in a meaningful way.

2.1 Requirements and specifications in performance testing

Performance requirements must be provided in a concrete, verifiable manner [VoW98]. This should be explicitly included in a requirements or specification document and might be provided in terms of throughput or response time, and might also include system availability requirements.

One of the most serious problems with performance testing is making sure that the stated requirements can actually be checked to see whether or not they are fulfilled [WeV00]. For instance, in functional testing, it seems to be useless to choose inputs with which it is entirely impossible to determine whether or not the output is correct. The same situation applies to performance testing. It is important to write requirements that are meaningful for the purpose of performance testing. It is quite easy to write a performance requirement for an ATM such as, one customer can finish a single transaction of withdrawing money from the machine in less than 25 seconds. Then it might be possible to show that the time used in most of the test cases is less than 25 seconds, while it only fails in one test case. Such a situation, however, cannot guarantee that the requirement has been satisfied. A more plausible piece of performance requirement should state that the time used in such a single transaction is less than 25 seconds when the server at host bank is run with an average workload. Assume that a benchmark has been established which can accurately reflect the average workload, it is then possible to test whether this requirement has been satisfied or not.

2.2 Characterizing the workload

In order to do the workload characterization, it is necessary to collect data for significant periods of time in production environment. This can help characterize the system workload, and then use these representative workloads to determine what the system performance will look like when it is run in production on significantly large workloads.

The workload characterization approach described by Alberto Avritzer and Joe Kondek [AKL02] is comprised of two steps that will be illustrated as follows.

The first step is to model the software system. Since most industrial software systems are usually too complex to handle all the possible characteristics, then modeling is necessary. The goal of this step is thus to establish a simplified version of the system in which the key parameters have been identified. It is essential that the model be as close enough to the real system as possible so that the data collected from it will realistically reflect the true system's behavior. Meanwhile, it shall be simple enough as it will then be feasible to collect the necessary data.

The second step is to collect data while the system is in operation after the system has been modeled, and key parameters identified. According to the paper [AKL02], this activity should usually be done for periods of two to twelve months. Following that, the data must be analyzed and a probability distribution should be determined. Although the input space, in theory, is quite enormous because of the non-uniform property of the frequency distribution, experience has shown that there are a relatively small number of inputs which actually occur during the period of data collection. The paper [AKL02] showed that it is quite common for only several thousand inputs to correspond to more than 99% of the probability mass associated with the input space. This means that a very accurate picture of the performance that the user of the system tends to see in the field can be drawn only through testing the relatively small number of inputs.

2.3 Developing performance test cases

After performing the workload characterization and determining what are the paramount system characteristics that require data collection, now we need to use that information to design performance test cases to reflect field production usage for the system. The following prescriptions were defined by Weyuker and Vokolos [WeV00]. One of the most interesting points in this list of prescriptions is that they also defined how to design performance test cases in case the detailed historical data is unavailable. Their by then situation was that a new platform has been purchased but not yet available; plus software has already been designed and written explicitly for the new hardware platform. The goal of such work is to determine whether there are likely to be performance problems once the hardware is delivered and the software is installed and running with the real customer base.

Typical steps to form performance test cases are as follows:

- identify the software processes that directly influence the overall performance of the system,
- for each process, determine the input parameters that will most significantly influence the performance of the system. It is important to limit the parameters to the essential ones so that the set of test cases selected will be of manageable size,

- determine realistic values for these parameters by collecting and analyzing existing usage data. These values should reflect desired usage scenarios, including both average and heavy workloads.
- if there are parameters for which historical usage data are not available, then estimate reasonable values based on such things as the requirements used to develop the system or experience gathered by using an earlier version of the system or similar systems.
- if, for a given parameter, the estimated values form a range, then select representative values from within this range that are likely to reveal useful information about the performance behavior of the system. Each selected value should then form a separate test case.

It is, however, important to recognize that this list cannot be treated as a precise preparation for test cases since every system is different.

3 Early performance testing with distributed application

Testing techniques are usually applied towards the end of a project. However, most researchers and practitioners agree that the most critical performance problems, as a quality of interest, depend upon decisions made in the very early stages of the development life cycle, such as architectural choices. Although iterative and incremental development has been widely promoted, the situation concerned with testing techniques has not been changed so much.

With the increasingly advance in distributed component technologies, such as J2EE and CORBA, distributed systems are no longer built from scratch [DPE04]. Modern distributed systems are often built on top of middlewares. As a result, when the architecture is defined, a certain part of the implementation of a class of distributed applications is already available. Then, it was argued that this enables performance testing to be successfully applied at such early stages.

The method proposed by Denaro, Polini and Emmerich [DPE04] is based upon the observation that the middleware used to build a distributed application often determines the overall performance of the application. However, they also noted that only the coupling between the middleware and the application architecture determines the actual performance. The same middleware may perform quite differently under the context of different applications. Based on such observation, architecture designs were proposed as a tool to derive application-specific performance test cases which can be executed on the early available middleware platform on which a distributed application is built. It then allows measurements of performance to be done in the very early stage of the development process.

3.1 Early testing of performance

The approach for early performance testing of distributed component-based applications consists of four phases [DPE04]:

- selection of the use-case scenarios relevant to performance, given a set of architecture designs,
- mapping of the selected use-cases to the actual deployment technology and platform,
- creation of stubs of components that are not available in the early stages of the development, but are needed to implement the use cases, and
- execution of the test.

The detailed contents in each phase are discussed in the following sub-sections.

3.1.1 Selecting performance use-cases

First of all, the design of functional test cases is entirely different from the case in performance testing as already indicated in the previous section. However, as for performance testing of distributed applications, the main parameters relating to it are much more complicated than that described before. Table 1 is excerpted from the paper [DPE04] to illustrate this point.

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

Table 1: Performance parameters [DPE04].

Apart from traditional concerns about workloads and physical resources, consideration about the middleware configuration is also highlighted in this table (in this case, it describes J2EE-based middleware). The last row of the table classifies

the relative interactions in distributed settings according to the place where they occur. This taxonomy is far from complete, however, it was believed that such a taxonomy of distributed interactions is key for using this approach. The next step is the definition of appropriate metrics to evaluate the performance relevance of the available use-cases according to the interactions that they trigger.

3.1.2 Mapping use-cases to middleware

At the early stage of development process, software architecture is generally defined at a very abstract level. It usually just describes the business logic and abstract many details of deployment platforms and technologies. From this point, it is necessary to understand how abstract use-cases are mapped to possible deployment technologies and platforms.

To facilitate the mapping from abstract use-cases to the concrete instances, software connectors might be a feasible solution as indicated [DPE04]. Software connectors mediate interactions among components. That is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required [MMP00]. According to the paper [MMP00], four major categories of connectors, *communication, coordination, conversion, and facilitation*, were identified. It was based on the services provided to interacting components. In addition, major connector types, *procedure call, data access, linkage, stream, event, arbitrator, adaptor, and distributor*, were also identified. Each connector type supports one or more interaction services. The architecturally relevant details of each connector type are captured by dimensions, and possibly, sub-dimensions. One dimension consists of a set of values. Connector species are created by choosing the appropriate dimensions and values for those dimensions from connector types. Figure 1 depicts the software connector classification framework which might provide a more descriptive illustration about the whole structure.

As a particular element of software architecture, software connector was studied to investigate the possibility of defining systematic mappings between architectures and middlewares. Well characterized software connectors may be associated with deployment topologies that preserve the properties of the original architecture [DPE04]. As indicated, however, further work is still required to understand many dimensions and species of software connectors and their relationships with the possible deployment platforms and technologies.

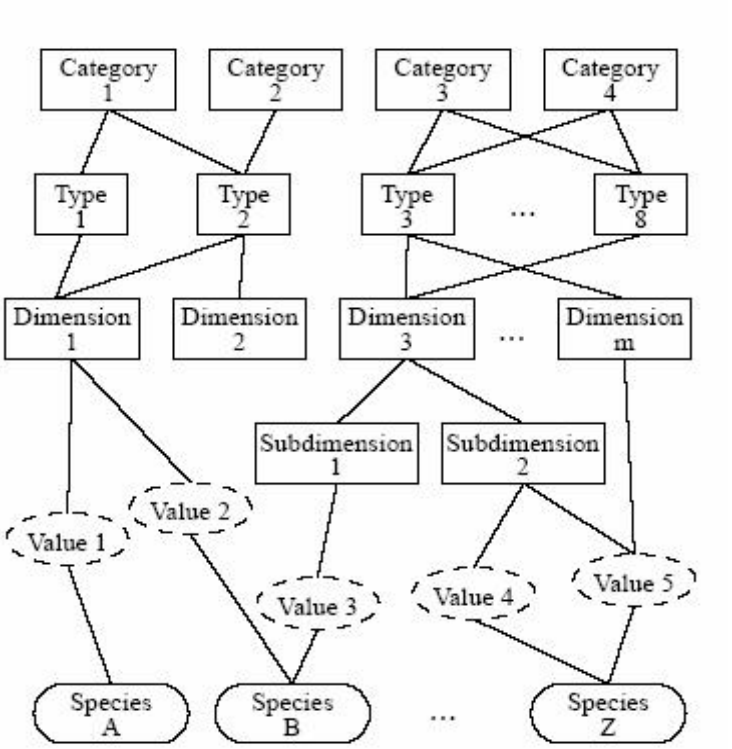

Figure 1: Software connector classification framework [MMP00].

3.1.3 Generating stubs

To actually implement the test cases, it needs to solve the problem that not all of the application components which participate in the use-cases are available in the early stages of development. Stubs should be used in place where the components miss. Stubs are fake versions of components that can be used instead of the corresponding components for instantiating the abstract use-cases. Stubs will only take care that the distributed interactions happen as specified and the other components are coherently exercised.

The main hypothesis of this approach is that performance measurements in the presence of the stubs are decent approximations of the actual performance of the final application [DPE04]. It results from the observation that the available components, for instance, middleware and databases, embed the software that mainly impact 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.

3.1.4 Executing the test

Building the support to test execution involves more technical problems provided scientific problems raised in the previous three sub-sections have been solved. In addition, several aspects, for example, deployment and implementation of workload generators, execution of measurements, can be automated.

4 Conclusion

In all, two software performance testing approaches were described in this paper. Workload characterization approach can be treated as a traditional performance testing approach that requires to carefully collecting a series of data in the production field and that can only be implemented at the end of the project. In contrast, early performance testing approach for distributed software applications seems to be more novel since it encourages to implement performance testing early in the development process, say, when the architecture is defined. Although it is still not a very mature approach and more researches need to be conducted upon it according to its advocators [DPE04], its future looks like to be promising since it allows to fix those performance problems as early as possible which is quite attractive.

Several other aspects also need to be discussed. First of all, there has been very little research published in the area of software performance testing. For example, with the search facility IEEE Xplore, if one enters *software performance testing* in the search field, there were only 3 results returned when this paper was written. Such a situation indicates that the field of software performance testing as a whole is only in its initial stage and needs much more emphasis in future. Secondly, the importance of requirements and specifications is discussed in this paper. The fact, however, is that usually no performance requirements are provided, which means that there is no precise way of determining whether or not the software performance is acceptable. Thirdly, a positive trend is that software performance, as an important quality, is increasingly punctuated during the development process. Smith and Williams [SmW02] proposed Software Performance Engineering (SPE) which is a systematic, quantitative approach to constructing software systems that meet performance objectives. It aids in tracking performance throughout the development process and prevents performance problems from emerging late in the life cycle.

References

AKL02 Avritzer A., Kondek J., Liu D., Weyuker E.J., Software performance testing based on workload characterization. *Proc. of the 3rd international workshop on software and performance*, Jul. 2002, pp. 17-24.

AvW04 Avritzer A., and Weyuker E.J., The role of modeling in the performance testing of E-commerce applications. *IEEE Transactions on software engineering*, 30, 12, Dec. 2004, pp. 1072-1083.

BCK03 Bass L., Clements P., Kazman R., *Software architecture in practice, second edition.* Addision Wesley, Apr. 2003.

DPE04 Denaro G., Polini A., Emmerich W., Early performance testing of distributed software applications. *Proc. of the 4th international workshop on software and performance,* 2004, pp. 94-103.

MMP00 Mehta N., Medvidovic N. and Phadke S., Towards a taxonomy of software connectors. *In proc. of the 22nd International conference on software engineering,* 2000, pp. 178-187.

Mus93 Musa J.D., Operational profiles in software reliability engineering. *IEEE Software,* 10, 2, Mar. 1993, pp. 14-32.

SmW02 Smith C.U. and Williams L.G., *Performance solutions: a practical guide to creating responsive, scalable software.* Boston, MA, Addision Wesley, 2002.

VoW98 Vokolos F.I., Weyuker E.J., Performance testing of software systems. *Proc. of the 1st international workshop on software and performance,* Oct. 1998, pp. 80-87.

WeV00 Weyuker E.J. and Vokolos F.I., Experience with performance testing of software systems: issues, an approach and a case study. *IEEE Transactions on Software Engineering,* 26, 12, Dec. 2000, pp. 1147-1156.