mBrace Agile Performance Testing
White paper

Version 2.2
03 July 2015
Author: Michael Kok

Inhoud

1 Introduction

Poor application performance can cause serious damage (loss of revenue, imago damage, costs of repair) to a business. Performance testing is the technique to reduce risks of poor application performance. Performance testing is commonly done through load and stress testing in the Implementation Phase only after the entire application is ready. Load and stress testing is complicated and expensive. Moreover the fact that it is done so late in the life cycle of the application has serious disadvantages. The cycle *Build – Test – Improve* takes too long. Repairing defects is inefficient this way and there is the chance of unexpected need for redesign, which can seriously delay delivery of the application.

At the same time application development faces increased pressure from the following requirements:

- Earlier insight in application quality including performance
- Earlier testing (begins in Construction)
- Speedup of delivering applications
- Continuous delivery and integration
- High release frequency (2 releases per day)

With conventional load and stress testing done in the Implementation phase, application development cannot meet these requirements. The above requirements need a method of performance testing that can be done much earlier while the application is still in its Construction phase. If we want to meet these requirements it is obvious that performance testing should shift from testing in the Implementation phase to the Construction phase of the application life cycle.

mBrace developed the Agile Method of performance testing on an entirely different paradigm applying transaction profiling and performance modelling. This eliminates the disadvantages and enables you to integrate performance testing with construction of applications the Agile way.

The tooling is installed in an arbitrary test environment with modest requirements. The test environment for instance does not need to have production like capacities. The developers can conduct the performance tests by themselves while building the transactions of the application. They start the test and execute one or more transactions that have been built. The tooling produces a profile of each transaction tested. Ref[1] shows more detail about transaction profiles the mBrace way. Next they can inspect the results and decide if the performance is OK. If not they can see where the best PIOs (Performance Optimisation Opportunities) are and start improving immediately until the transaction meets the requirements. Software can be delivered under a new definition of Done that now includes performance.

In this white paper we outline the way Agile Performance Testing is done. In the next sections we have a look at the nature of performance testing and the differences between conventional and model based performance testing and then show how it is done.

2 The nature of performance testing

Performance testing is done in the first place to reduce the risk of poor performance for the end-user, since this can cause severe damage to a business. Hence performance testing should be governed by a risk assessment. But what does the risk looks alike?

In order to achieve acceptable user experience we have to deal with three main areas of risk that are revealed in the following figure.

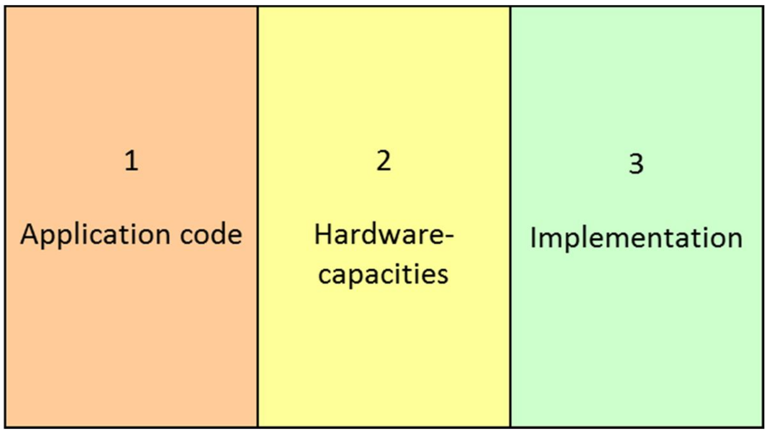


Figure 1. Performance risk areas of an application

1. Application code.
   The software of the application should be efficient qua time and resource consumption. Small innocent problems may cause harmful defects. Loops must be as short as possible, system services must be used in the right way, database locks and mutexes must be involved as short as possible, memory leaks must be avoided etc.
   Securing the performance potential of the application code is commonly done in the Acceptance test just prior to go live. This often happens several months after the code was created. Repair may be done by another programmer or the original programmer may already have forgotten many of the considerations underpinning his code. Ideally repair should be done in the Construction phase of application development as soon after the code was created as possible. This can be done component by component while steadily completing the application as a whole.

2. Hardware and software capacities.
   Many performance problems are caused by lack of capacity on one or more resources. When implementing the application in a production environment there must be sufficient capacities for all resources, hardware and software. Hardware resources include CPU's, disks, memory, networks etc. Software resources include session pools, connection pools, heaps, garbage collectors, locks and single threaded software modules. Capacity supplies must be such that utilisation percentages of these resources must be sufficiently low at the target volume of the application in order to avoid delays caused by queuing.
   Securing the balance between supply and demand of capacities is now done commonly in the Acceptance performance test, but can be done earlier at the end of the Construction phase immediately after building the entire application is completed.

   Model based performance testing is much more efficient for this. Conventional testing stumbles from bottleneck to bottleneck. In contrast, model based testing shows all bottlenecks after one test cycle together with accurate quantification. If there are capacity issues on software resources, conventional testing will show that an issue may exist without revealing which resource is the cause. The performance model helps to pinpoint the software resource and provides the parameter setting.

3. Implementation
   At implementation all configuration and parameter settings must be optimal and all hardware and software must work functionally correct. Testing this is done commonly in the Acceptance phase, but should be done earlier at the end of the Implementation phase.
   Proper functioning of hardware, e.g. does the load balancer correctly distribute the traffic over four servers, should be physically tested with a load test. Hence (if the risk assessment indicates so) Implementation testing requires conventional load testing and this should be done no earlier than after the infrastructure and the platforms are prepared for commissioning the first release of an application. Whether this has to be done anyway depends on the risk assessment and following considerations. Do we release a new built on an existing infrastructure that has proven to be OK? Then we have a low risk and the Implementation test could be skipped. If an extra

piece of functionality is added to a large application, we need to test the code, guard the capacities but the risk that the infrastructure all of a sudden causes new problems is low. Hence the implementation test could be skipped.

There are not many cases where Implementation testing is necessary and when it is done it can be done with a small selection of the application functionality saving much scripting and test effort.

A risk assessment that forms the basis for decision making on conducting performance tests yes or no, should address each of the above three areas.

With Agile performance testing the point of gravity of performance testing shifts to the earlier stages of the application development process. But as was shown above not all testing is shifted upfront towards the Construction phase of the application. The Code testing shifts completely upfront to the Construction Phase. Capacity optimisation shifts back to the end of the Construction phase and Implementation testing can be done after the preparations for commissioning are complete. Acceptance testing shrinks to a demo to the end-users.

3 Model based testing vs. conventional testing

The conventional method of performance testing throws load at a real system. When the load is increased the response times extend until they are not acceptable. Then we know the max load and can decide if that is enough to commission the application, but we do not gain much more insight for improvement. We do not obtain response time breakdowns so we do not know much about where the transactions spend their time.
The mBrace method for model based performance testing is based on transaction profiling. The next figure shows the profile of one transaction (For the sake of simplicity only one transaction profile is shown. Commonly the figure shows a bunch of them.) executed on a multitier infrastructure with client PC’s, web servers, application servers and database servers, all interconnected by networks.

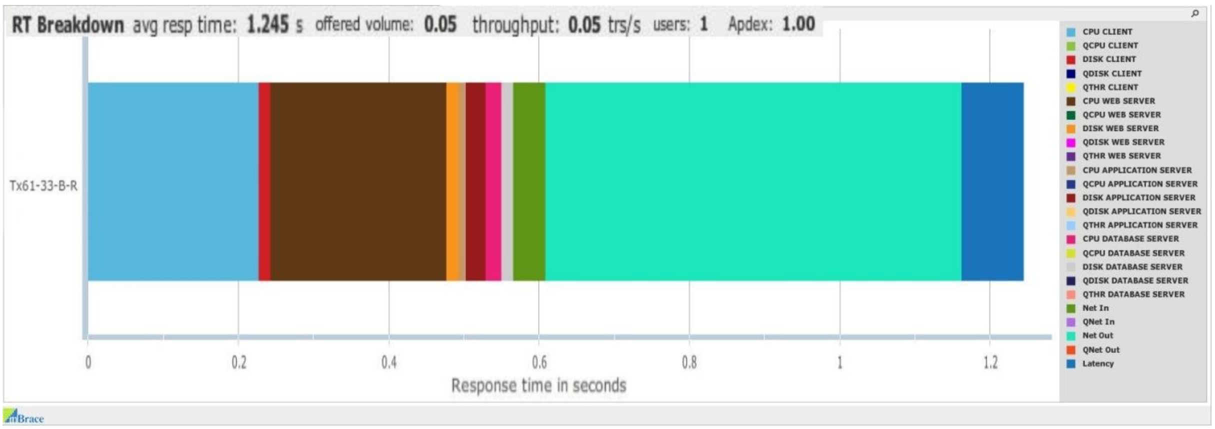


Figure 2. Transaction profile

The transaction profiles reflect the Performance-DNA of the application software and show the single user response time (1.2 seconds) plus a breakdown of it. E.g. the large green part reflects the time spent in the outbound part of the network (Net Out). The legend to the right explains the colours of the breakdown. The profiles make it clear where the transactions spend or waste their time. The transaction profile reveals the performance potential of the transaction at a glance and by showing the PIOs (Performance Improvement Opportunities) plays a crucial role in evaluating performance. In this example it is obvious that the green part (time needed for outbound network traffic of the transaction) takes most of the time. Hence outbound network IO is the first candidate for improvement.

When the transaction volume is input to the model the multiuser response time can be predicted for the target transaction volume. The result is shown in a similar figure that shows a response time and breakdown that are extended by time consuming components for queuing for loaded resources. This enables us to do load tests and stress tests within the model without throwing real load to an application in some test environment.

Next we show an example of a *Build – Test – Improve* cycle.

4 Apdex Index: judge good or bad

For deciding about good or bad performance you might (at your own discretion) use the Apdex Index. mBrace supports the Apdex Index in its tooling.

The Apdex Index requires norms that are set by user representatives. This is what Wikipedia tells about Apdex: "Apdex (Application Performance Index) is an open standard developed by an alliance of companies, the Apdex Alliance. It defines a standard method for reporting and comparing the performance of software applications in computing. Its purpose is to convert measurements into insights about user satisfaction, by specifying a uniform way to analyze and report on the degree to which measured performance meets user expectations."

Based on the user's opinions on acceptable performance a low limit and a high limit are set. When a transaction has a response time below the low limit its count for "excellent" is increased by 1. When a transaction has a response time above the high limit it has unacceptable performance and the count for "excellent" is not increased. When a transaction has a response time between the low and the high limit the count for "excellent" is increased by only 0.5. The resulting score is divided by the total number of transactions yielding a figure between 0 and 1, the Apdex Index. A score of 1 means that the application fully meets the requirements of the user and has excellent performance. A score of 0 means that no transaction meets the requirements of the users and the application performs unacceptable. E.g. a 0.6 score means barely sufficient etc.

In the example we are showing in the next sections the limits are set to 2, respectively 8 seconds.

After each test cycle / measurement the Apdex Index is determined guiding the developer for deciding to further optimise the code or not.

5 Sprint 01 / Transaction 01

The picture below shows two instances of profiles from the same transaction. The horizontal coloured bars represent the profiles. The length of the bar indicates the total end to end response time of the transaction. The coloured parts correspond with the times spent on the hardware resources of the multitier infrastructure. The upper bar of the two (response time 8.1 seconds) reflects the performance of the transaction in the test environment. The lower bar (response time 5.6 seconds) is a projection of the performance of the transaction when executed in the production environment. Obviously the hardware of the production environment is faster than the test environment.

Both bars relate to the transaction before optimization.

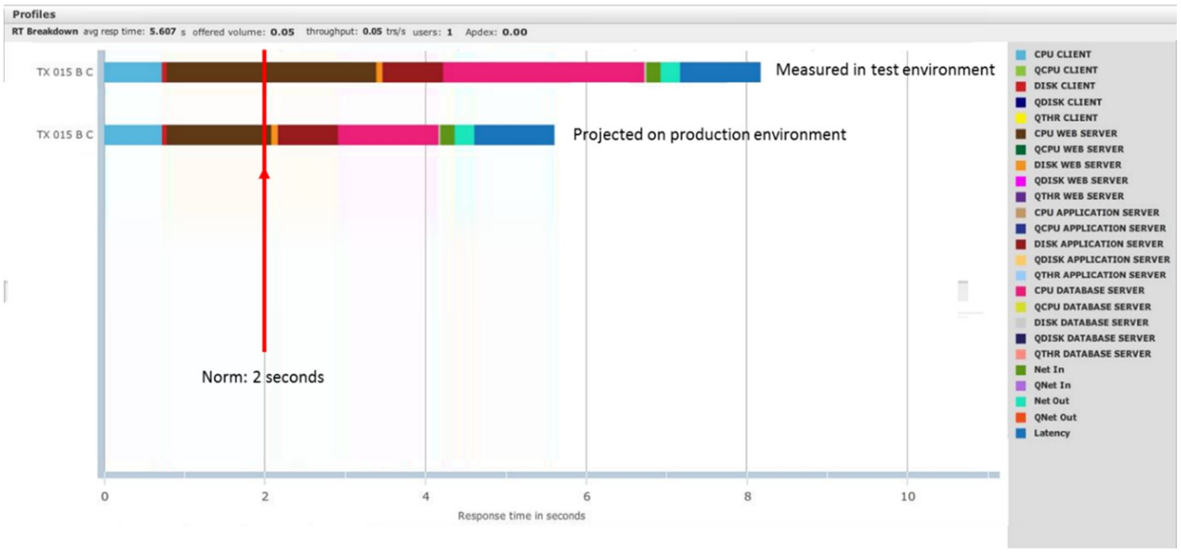


Figure 3.

After optimization, the resulting profile, projected on the production environment only, looks like the next picture shows (picture on the same scale!)

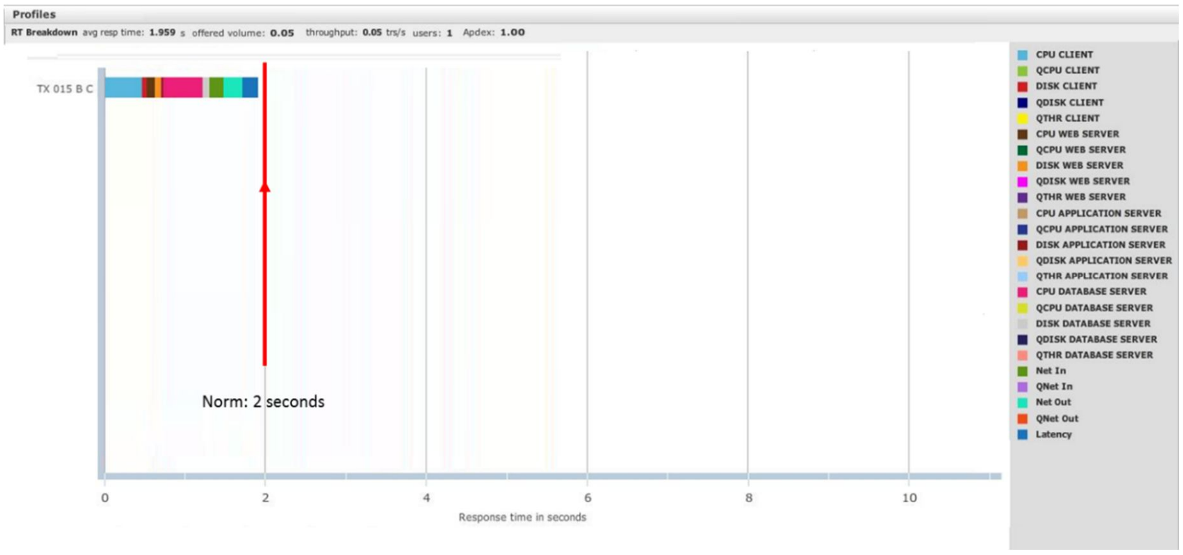


Figure 4.

In order to make the projection onto the production environment, of course the hardware characteristics of both environments have been fed into the model.

Mind the scale of response time in the chart! This is the same transaction:

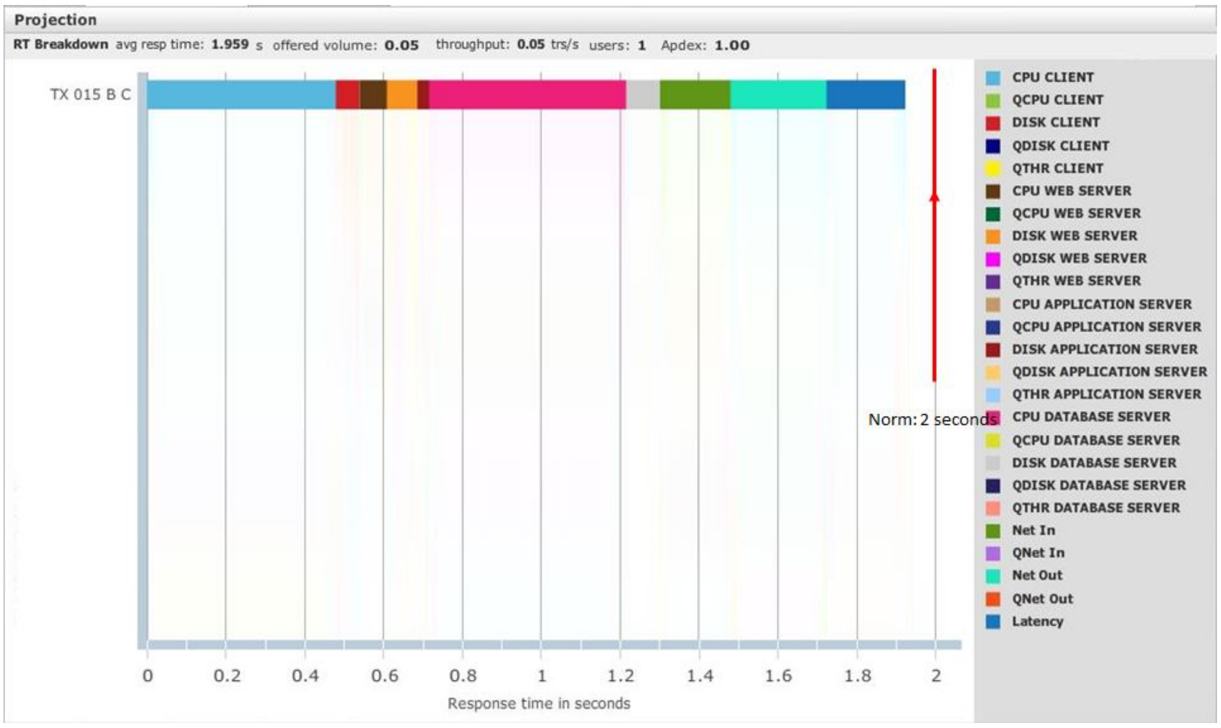


Figure 5.

6 Sprint 01 before and after optimization

6.1 Sprint 01 before optimization

You can include as many transactions in a performance test as you like. You could for instance test and optimize a complete sprint with four transactions.

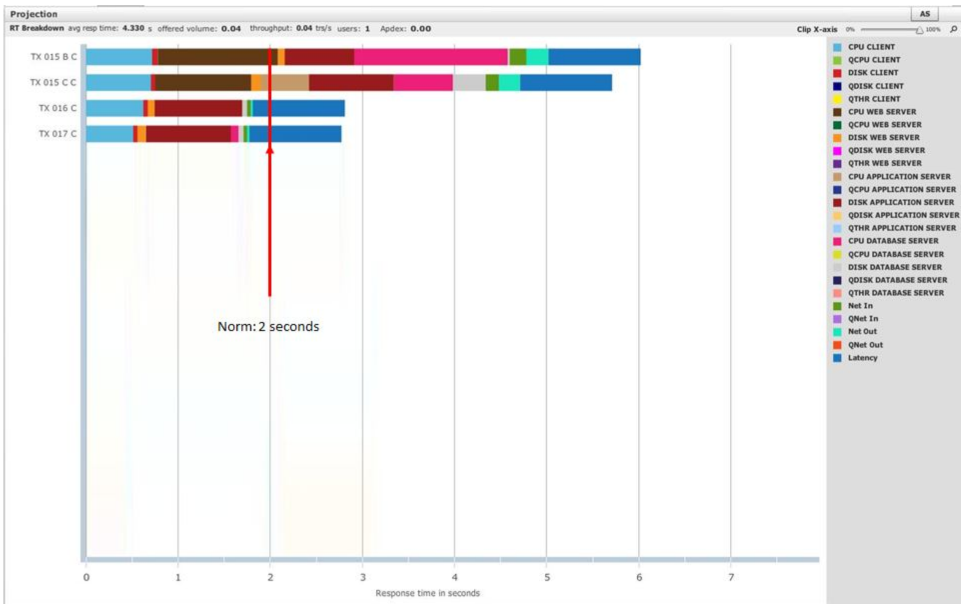


Figure 6.

6.2 Sprint 01 after optimization

Most of the transactions are below 2 seconds now. Only one is still excessive. However the Apdex Index displays 0.95, which is good performance. Hence in this case the Scrum Master may decide to accept it for the time being, or shift more work to the backlog.

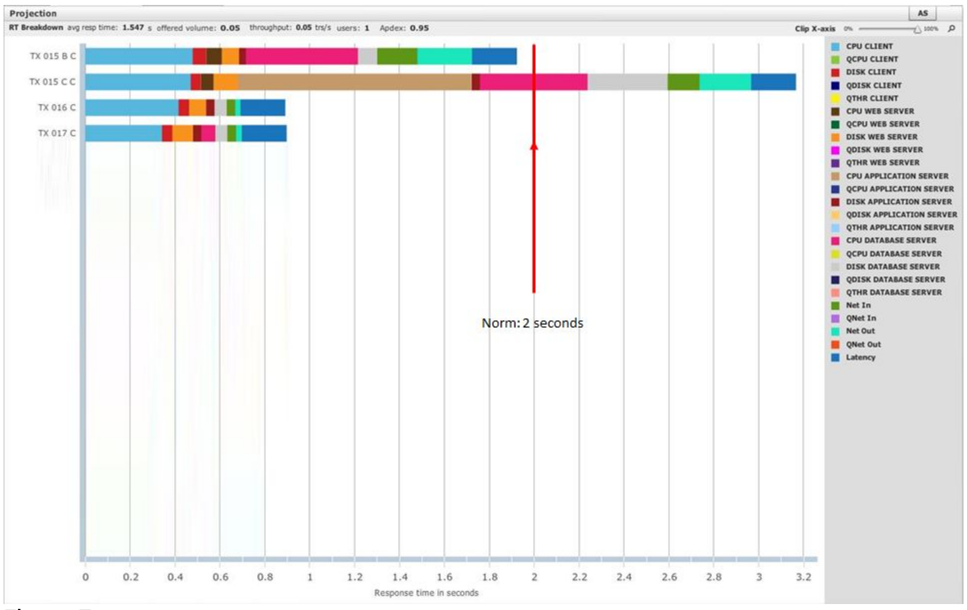
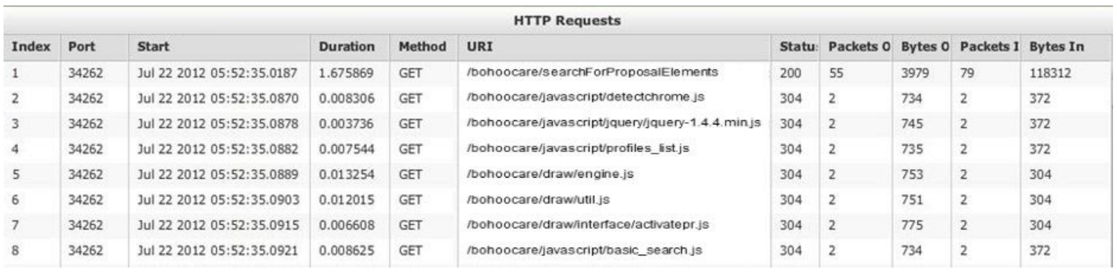


Figure 7.

7 Drill down

Transaction types *Tx 015 B C and Tx 015 C C* in Sprint01 have response times up to 6 seconds of which 2 seconds at the Web server and had to be reworked to optimise its performance. One of the drill down features shows details of the web requests for each transaction type and helps to find the best way to optimise. In this example the drill down feature reveals that 7 http requests are done in transaction type *Tx015 B C*. See figure below that shows their names, response times and amount of data transferred over the network.



HTTP Requests

Index	Port	Start	Duration	Method	URI	Statu:	Packets O	Bytes O	Packets I	Bytes In
1	34262	Jul 22 2012 05:52:35.0187	1.675869	GET	/bohoocare/searchForProposalElements	200	55	3979	79	118312
2	34262	Jul 22 2012 05:52:35.0870	0.008306	GET	/bohoocare/javascript/detectchrome.js	304	2	734	2	372
3	34262	Jul 22 2012 05:52:35.0878	0.003736	GET	/bohoocare/javascript/jquery/jquery-1.4.4.min.js	304	2	745	2	372
4	34262	Jul 22 2012 05:52:35.0882	0.007544	GET	/bohoocare/javascript/profiles_list.js	304	2	735	2	372
5	34262	Jul 22 2012 05:52:35.0889	0.013254	GET	/bohoocare/draw/engine.js	304	2	753	2	304
6	34262	Jul 22 2012 05:52:35.0903	0.012015	GET	/bohoocare/draw/util.js	304	2	751	2	304
7	34262	Jul 22 2012 05:52:35.0915	0.006608	GET	/bohoocare/draw/interface/activatepr.js	304	2	775	2	304
8	34262	Jul 22 2012 05:52:35.0921	0.008625	GET	/bohoocare/javascript/basic_search.js	304	2	734	2	372

Table 1. Drill down details of transaction type *Tx015 B C*: its web requests

The first http request in the list named *searchForProposalElements* shows the longest response time, 1.675869 sec, it sent 3,979 bytes out and received 118,312 bytes. Hence the decision was made to take another look at this request in order to optimise the performance of transaction type *Tx015 B C*. Upfront we know that the response time can be shortened with 1.675869 sec at max. Hence upfront we also know that this may solve the problem a great deal, but not completely. The result was a new request *quickSearchForProposalElements* with a response time of 0. 175754 that shortened the response time with 1.5 seconds. Together with other improvements the response time was brought below 2.0 seconds.

8 Whole application after 5 sprints – Time behavior

After all 5 sprints have finalized the profiles of the complete application with 20 transactions may be shown for viewing the total improvement.

8.1 Before optimization

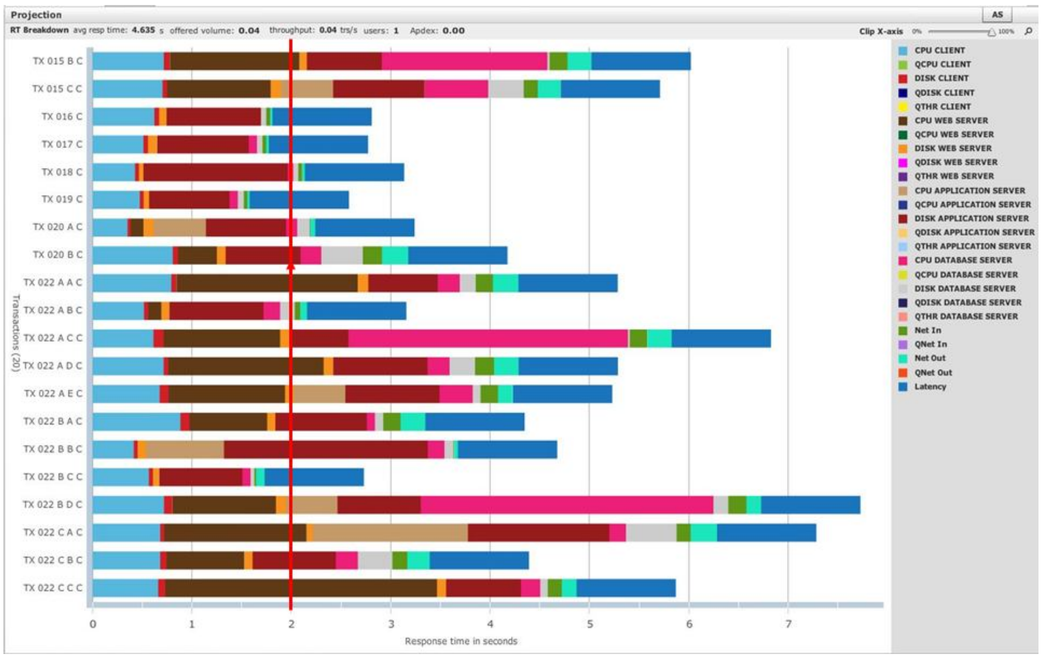


Figure 8. All transactions initially exceeded the norm firmly.

8.2 After optimization

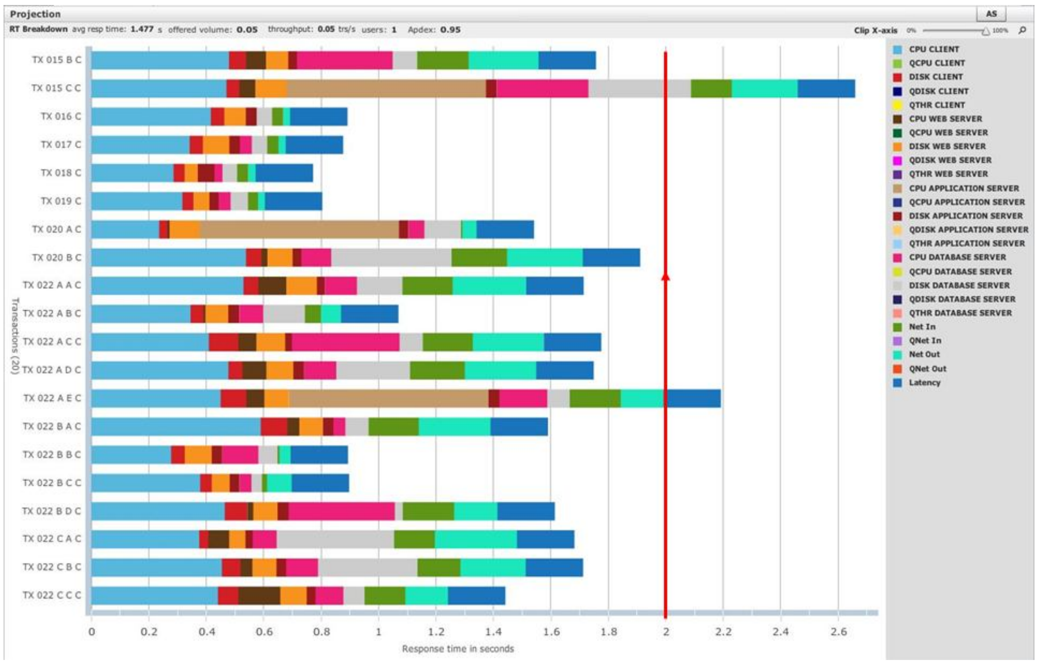


Figure 9. Two transactions still exceed the norm of 2 seconds. Nevertheless the application as whole scores an Apdex Index of 0.95, which is excellent.

9 Whole application - resource usage

9.1 *Before optimisation*

When construction of the application has completed it makes sense to inspect its resource usage. The upper part of the next chart unveils the utilisations on the hardware resources. The utilisations should not be too high in order to avoid excessive queuing. The next two figures show the impact of the improvements on resource utilization.

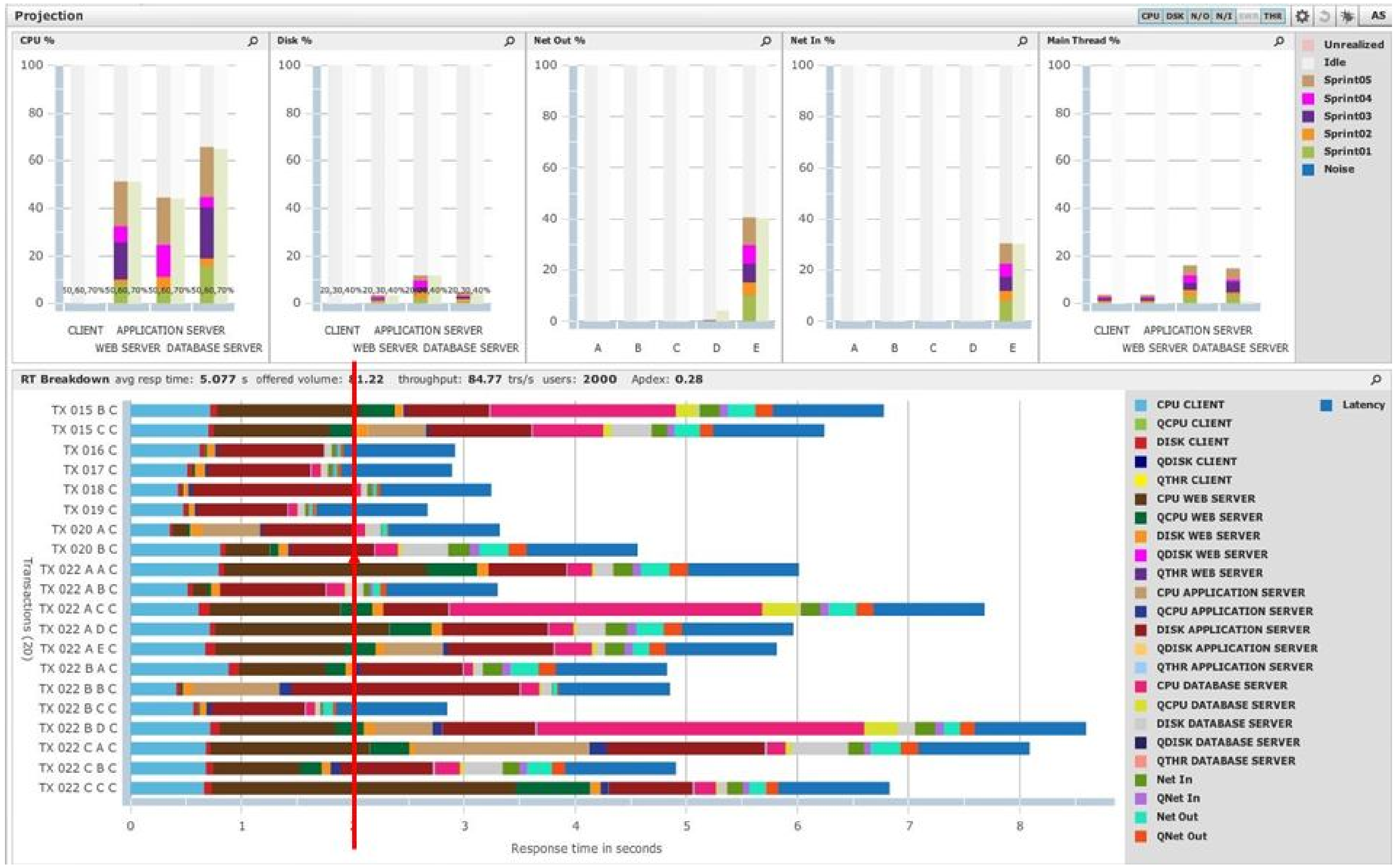


Figure 10.

9.2 *After optimisation*

The next figure shows that not only the time behavior of the application has improved, but also that this has considerable positive impact on resource usage. Hardware can be scaled down.

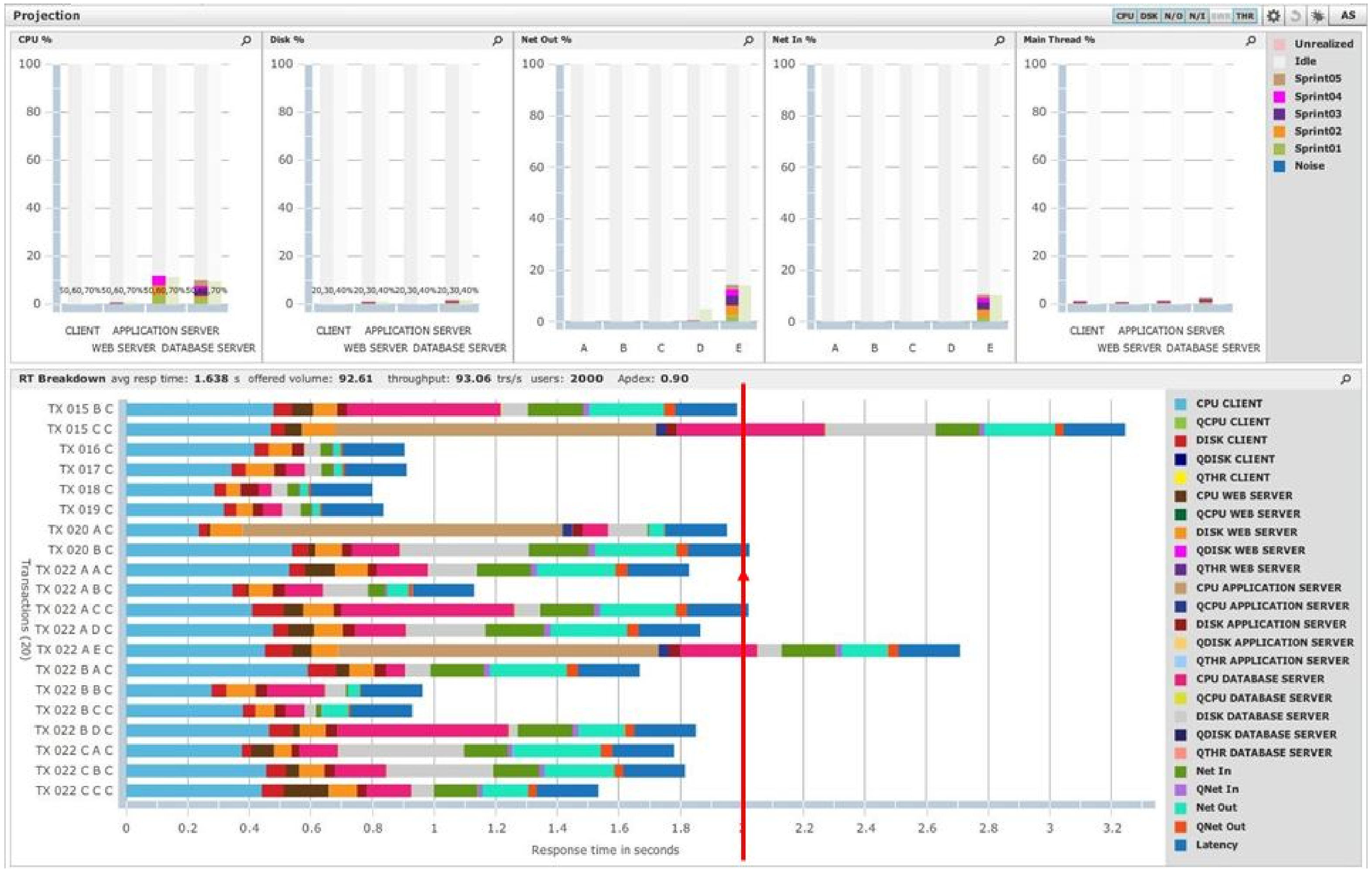


Figure 11.

10 Concluding remarks

Two capabilities of the mBrace Agile Performance Testing method make it especially suitable for testing in Agile application development:

- Transaction profiling
- Application performance modelling

Together these two capabilities provide a method based on an entirely different paradigm.

Multitier transaction profiles (Performance DNA of the application) provide insight in the quality of the code.
The performance model unveils the impact of that code quality on overall performance and capacity need.

Performance modelling makes it possible to project transaction profiles obtained in an arbitrary test environment and project them on the production environment. This enables us to optimise the performance of an application much earlier in its lifecycle by integrating performance testing within the Construction phase of application development.

The mBrace Data Collection Protocol enables the developers to efficiently test the code with a simple push on the button. No specialised performance testers are needed. With the results the developers oversee at a glance the performance of the transactions they build, if optimisation is necessary and where the PIOs (Performance Improvement Opportunities) are. This (together with the mBrace method for Regression Performance testing, see the separate white paper) enables a test method with the following benefits:

- Earlier insight in application quality including performance
  Is given transaction by transaction while the application is built.
- Earlier testing
  Performance testing is done during Construction instead of during Implementation.
- Speedup of delivering applications
  The cycle *build – test - optimise / repair* is much shorter. Optimisation or reparation takes much less time during construction, than after several months in the Implementation phase.
- Continuous delivery and integration
  Functionality to be released is not only functionally tested, but also tested and optimised for performance, immediately ready for release.
- High release frequency (2 releases per day)
  The short test and optimisation cycle enables the developers to deliver code that has been tested and improved for performance. The definition of done is complete, so that the software is ready for deployment enabling a high release frequency.